UNIVERSITY OF OSLO
Department of Informatics

Design and Implementation of a Rudimentary WirelessHART Network

Anders Asperheim

Rune V. Sjøen

Kaja F. L. Skaar

August 9, 2012

Preface

This report is the final result of a 60 point masters project completed at the University of Oslo, Institute of Informatics. The work has been performed by the students Anders Asperheim, Kaja F. L. Skaar and Rune V. Sjøen in the time period 2010-2012. The project has been supervised by professor Stein Gjessing. As an external supervisor we have also had contact with Niels Aakvaag who works at SINTEF, which is the largest research organization in Scandinavia and is located in Norway.

We have continued on a project started in 2009 with the goal of providing a full fledged implementation of the WirelessHART protocol on micro controllers from Atmel. Our work is based on this project and our primary goal was to finish implementing the design of the link layer and then continue to provide a basic implementation of a Network Manager and the network layer.

The project has not been without challenges but we feel that we have overcome most of them and are now much closer to a fully functioning implementation of WirelessHART.

The background research for this project has mainly been performed by studying the WirelessHART standard and literature regarding related work in the field.

We would like to thank Stein for guiding us through the project work and offering constructive support and criticism during our regular meetings. In addition we would like to thank Niels and all our fellow students in the Network and Distributed Systems lab at the Institute of Informatics for their support and assistance.

Summary

This report provides an overview over our efforts in continuing the work started by two previous students with the goal of implementing a full WirelessHART network stack on Atmel wireless sensor nodes. This is a collaborative project between the University of Oslo, Statoil and SINTEF.

This report provides the reader with an introduction to the available protocols and technologies in use in the world of wireless sensor networks.

After giving the reader an introduction to wireless sensor networks and the underlying physical and link layer characteristics of IEEE 802.15.4 we explore WirelessHART in more detail along with an analysis of the implemented code base and how it performs in our test scenarios.

After carefully reviewing the implementation and final code base, it is clear that we have made advances in this area. In particular, our approach to the implementation of a WirelessHART Gateway was found to result in measurable improvements to the operation of the network and the continued effort to provide a full-fledged implementation of a WirelessHART Network. Our methods to address the protocol stack in the WirelessHART Field Devices resulted in an increase in stability and several new features which provide yet another step in the right direction.

It also seems evident that although our work has resulted in important updates and improvements in this project, we have not exhausted this area of research. More work should be performed to explore this area further in order to further improve and extend the implementation of both the WirelessHART Field Devices and the WirelessHART Gateway.

Contents

Chapter 1

Introduction

In this chapter we provide the reader with some background on the project. The problem definition will be explored in detail along with an introduction of the key assumptions and limitations and the reason why we consider this an interesting area of development, in addition to providing an introduction of the involved parties. A description of how we originally planned to perform time management within the project is presented along with the report outline and structure. At the end of this chapter, we discuss and provide an overview of relevant related work that has already been performed in this area.

1.1 Background

The HART protocol (Highway Addressable Remote Transducer) is a digital industrial communication automation protocol. It is widely deployed and accepted in the process industry. HART was originally developed by Rosemount Inc., but is now governed by the HART Communication Foundation (HCF). The protocol is a popular communication protocol and is installed in over 30 million devices around the world[4].

Wireless networks have existed since they were introduced with IEEE 802.11[5] towards the end of the 1990s, and since then several amendments have been made allowing for a higher bandwidth and lower latency. Later

the IEEE 802.15 series of standards[15] introduced specifications for Wireless Personal Area Networks (WPAN). One of these task groups are solely focused on Low Rate WPANs and is named 802.15.4. The key aspects of LR-WPANs are long battery life and very low complexity. During the later years, interest and effort have been made towards merging these two concepts, thus making the already widely adopted HART protocol utilize a wireless medium for communication. The primary motivations for this are both operational as well as financial since the introduction of WirelessHART into industrial applications could greatly reduce the deployment and maintenance costs for sensor networks.

In 2007 the WirelessHART standard was born, designed by HCF, describing how to implement a wireless sensor network based on the IEEE 802.15.4 standard and using it to carry HART traffic. A more detailed look at protocols for wireless sensor networks will be provided in chapter 2.

There are several manufacturers and resellers of WirelessHART-capable devices in the market. After WirelessHART was standardized in 2007, DUST Networks established themselves as the only vendor in 2008. During the later years, several other companies have made their way into the WirelessHART market. These companies include Software Technologies Group (STG), Texas Instruments (TI) and Atmel. The main differences between hardware from the vendors is the physical characteristics of the micro controllers and the radio sensitivity. While DUST Networks provide a protocol stack and TI also is working in this area, per today, there is still no open source protocol stack available for WirelessHART and we believe that an open stack would provide a significant advantage for future development of the WirelessHART standard.

1.2 Problem Definition

The overall goal of this project is to implement the minimum requirements for a working WirelessHART [29] network running on the hardware provided by Atmel in addition to a rudimentary Network Manager. This

project is a partnership between the University of Oslo, SINTEF and Statoil and we will continue the work started by two other master students in 2009. The previous work provides a partial implementation of the WirelessHART link layer and is explained in more detail in chapter 5.

1.3 Key Assumptions and Limitations

Throughout this project, we have been limited to the hardware provided by the Atmel corporation and the constraints that follow this hardware, we assume that the physical characteristics of these devices meet the minimum requirements of the WirelessHART standard to such a degree that a complete stack can be fully implemented. We do not have any external dependencies during this project and the risks of this project have been analyzed and a more detailed overview of the risks is provided in appendix C. In terms of resource limitations we are limited to a time period of 18 months and a total of 3 man-years of work.

1.4 Project Parties

The two main involved parties in this project were Statoil and Atmel. Below is a short introduction to these companies.

1.4.1 Statoil

Statoil is an energy company that was founded in Norway. With numerous oil and gas fields and refineries inside and outside of Norway, Statoil is an active user of sensor networks and already has a lot of experience with HART, the predecessor of WirelessHART. HART, however, is a standard for wired sensor networks and the price per meter of installed cable is quite high, making wired networks less than optimal. Statoil is therefore exploring alternatives to reduce cost and WirelessHART is a good candidate as it could potentially reduce the installation and maintenance costs significantly.

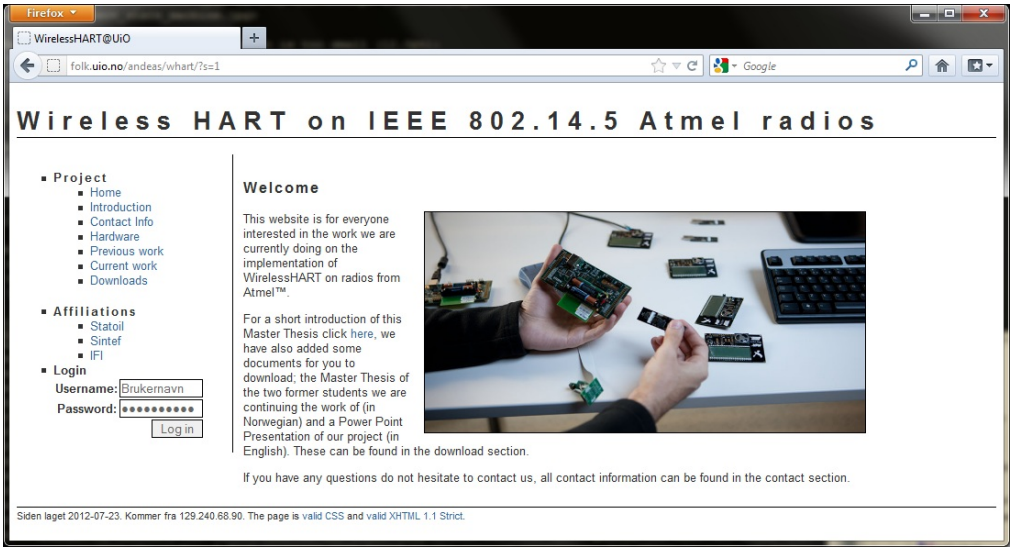


Figure 1.1: Our Project Website

1.4.2 Atmel

Atmel is a company that manufactures electronic circuits and micro controllers. Because the work we were picking up was programmed specifically for Atmel micro controllers, we continued working with these throughout the project.

In order to receive hardware and support from Atmel we had to enroll in the Atmel AVR University Program. As a prerequisite for joining this program we needed a project website[17] (Figure 1.1) with running updates on our progress. Atmel has also provided us with the hardware required to perform this project.

Atmel Norway is a subdivision of Atmel which is located in Trondheim, Norway. During the initial phase of our project we were in contact with the office in Trondheim and they were most helpful in routing us further into the Atmel support system and helped us get in contact with the proper instances at the Atmel office in the United States to answer some of our questions.

1.5 Project Baseline

Our implementation of WirelessHART is based on the thesis "WirelessHART - Gjennomgang og implementering" ("WirelessHART - Review and implementation") [41] by two former students at the University of Oslo Håvard Tegelsrud and Jørgen Frøysadal in May 2010.

In their thesis the two students reviewed WirelessHART and partly implemented it on Atmel hardware. Due to the restricted time they only managed to implement an almost complete Logical Link Control sub layer (LLC sub-layer) on the Link layer. In addition to this they also designed and partially implemented the link scheduler and the send and receive functionality on the MAC sub-layer. A more thorough review of the existing code and design will be provided in chapter 5.

Our work is based on the work of Tegelsrud and Frøysadal and our implementation will be a continuation of what they created. Our first goal was to finish implementing their design of the Data Link Layer and then continue to provide a basic design and initial implementation of a Network Manager and Network Layer.

During our design phase it became apparent to us that we started out on a lower level than anticipated and some of the functionality described in [41] were only in the design stage. This resulted in having to re-prioritize the functional requirements and focus on the high priority issues involved in completing the WirelessHART standard on the link layer. These changes are reflected in the functional requirement specifications of section 6.2.

1.6 Project Planning

Before starting the project, we discussed how to manage the project in order to spend time optimally. We decided that the best approach would be to start by getting to know the WirelessHART specification, and once we had a basic understanding of its operation, move on to test and evaluate the implementation provided. We would then move on to implement the missing features on the link layer described in section 6.2.

After the basic implementation of the data link layer on the wireless nodes were in place, we would move on to designing and implementing a basic Network Manager. The details on how to provide an interface between the Network Manager and the rest of the wireless network were not clear at the time. Throughout the project we wanted to test and evaluate our progress continuously as opposed to have fixed development/testing cycles.

When planning how to do time management within the project, we used Unified Process (UP) as a starting point. A more detailed description of UP can be found in appendix D. A description of how the project management turned out and the division of time for each milestone, is presented in section 8.5.

1.7 Report Outline

During the course of this report we will present the various stages of the project, we start by providing the reader with a description of the background and project parties in addition to a rundown of the previous work performed in the field of wireless sensor networks. In addition we provide detailed information about the previous work performed on this project and the starting point for our project.

We then move on to describing in detail the concept and various architectures of wireless networks and how they compare to their wired counterparts, after which we go in depth on the WirelessHART standard and how it compares to other protocols.

After this we continue on by providing a detailed description of the environment in which we have developed this project including the hardware packages, the software packages and the decision process involved in choosing which products to go for.

The design phase is then explained where both the functional and non-functional requirements are defined in addition to design proposals and our thoughts and decisions on how to implement the various modules. Information about how we planned to provide quality assurance, maintainability

and testing the code base, in order to make it easy to keep the project going after we finish, will also be provided.

In the implementation phase we provide detailed information on which steps we have taken throughout development and how the project has been implemented. We discuss which part of the implementations diverge from the proposed design and the reasoning behind this.

Finally we evaluate our implementation and make an effort to prove that the implementation works as defined in the WirelessHART standard and the intended design after which we will present our conclusions and suggestions to where to take the project from here.

1.8 Related Work

In this section we will describe some previous work that has been done on WirelessHART and we will try to utilize this knowledge when implementing our own WirelessHART network.

1.8.1 When HART goes wireless: Understanding and implementing the WirelessHART standard

In this article[31] Kim, A.N. et. al. map out and give a general overview over the challenges involved in implementing WirelessHART on the MAC layer in addition to implementation of a Network Manager. The article describes most of the common challenges we encountered during the course of our project. Challenges mentioned are timing limitations defined in the WirelessHART standard, how the protocol maps onto the standard OSI model and a short summary of the Network Manager implementation and the protocols on the network layer. The article also mentions the transport layer but this was not relevant for us at this particular stage of development.

1.8.2 WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control

This article[39] makes an effort to describe some challenges encountered during implementation of WirelessHART. The article is separated into three main parts, an introduction to WirelessHART, a study of some of the challenges in WirelessHART and some experiences and lessons learned during the implementation. The third part on some lessons learned and how some of these challenges can be solved is the most relevant part to this project, some examples are how to manage timing requirements, a rundown of the state machine and an introduction to the design of the network layer.

1.8.3 Comparison of the IEEE 802.11, 802.15.1, 802.15.4 and 802.15.6 wireless standards

This paper by Jan Magne Tjensvold is focused on describing the architectural differences between various wireless standards. While the differences on the physical layer like the frequency, modulation and coding schemes is not directly relevant for this project it makes an interesting read when attempting to get an overview over the vast amount of wireless standards.

1.8.4 WirelessHART Network Manager

Throughout the master project described in this thesis[22] key concerns and an analysis has been performed in regard to the design and implementation of a WirelessHART Network Manager. In the conclusion, it becomes clear that the author, Sanchéz, J.H., believes that a lack of a standardized design, architecture and description of the Network Manager could discourage the future development of the WirelessHART standard. Several key elements in designing a functional and easily extendible Network Manager are addressed including a specification of software requirements. We believe that this thesis may serve as a useful tool and reference material when moving forward to implementing a fully functional Network

Manager.

1.8.5 Design & Implementation of Time Synchronization for Real-Time WirelessHART network

This master thesis[42] written by Thamer Alyass and Mangalarapu Chaitanya Kumar at the Jönköping Institute of Technology deals with the issues of time synchronization in WirelessHART. They point out the drawbacks of the Time Synchronization and Channel Hopping (TSCH) technique the WirelessHART standard defines. Their goal was to extend it with the Flooding Time Synchronization Protocol (FTSP). The main drawback of TSCH described is time stamping errors which occur because of the time the interrupt handling, encoding and decoding takes. Some of the advantages of using FTSP over TSCH mentioned in the thesis are MAC layer time stamping, jitter reducing techniques and the elimination of the time stamping errors of TSCH. This thesis is relevant to our work, as time synchronization is a vital part of a functioning network, and the thesis highlights limitations with TSCH.

1.8.6 A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks

In this article[36] the authors mainly propose a classification scheme for mobile ad-hoc networks with emphasis on a set of parameters said to be generic in classifying such networks. They conclude that no particular algorithm will be able to suit all scenarios, but they supply a list classified after their own classification schemes that make it relatively clear what application the authors see as the most beneficial scenario for each of the reviewed algorithms. The article is interesting as several of the parameters in their classification scheme apply for infra-structured networks and it gives a differentiated view of wireless network routing.

1.9 Chapter Summary

WirelessHART is a fairly new contribution to the wireless world. it was standardized in 2007, and during the past years several suppliers for applicable hardware have emerged. In this chapter we have provided the reader with some background history of WirelessHART in addition to an introduction to research and effort in the same area. Our key assumptions during this project have primarily been that the project aims to provide a functional implementation on certain pieces of hardware from Atmel. The starting point for our project was somewhat lower than first expected and this has been explored and described in more detail in chapter 5. We have provided the reader an introduction to the involved parties, how we planned our project and a report outline that describes how this report is structured.

Chapter 2

Wireless Sensor Networks

Throughout this chapter we provide an introduction to the available methods of medium access including Carrier Sense Multiple Access (CSMA) and Time Division Multiple Access (TDMA), and evaluate them against each other as these two are commonly used in consumer wireless technology (e.g. GSM, WirelessHART, Wireless LAN and 3G). A commonly used analogy when explaining these two methods is that CSMA is when people speak different languages and TDMA is when people speak in different turns. We will take a closer look at this analogy later.

After describing methods of medium access we provide some information describing the various available standards and protocols in the world of wireless networks and how WirelessHART compares against them. We also explain how WirelessHART differs from 802.3 and 802.11 networks from both a functional and organizational point of view.

In addition to this we explain the subject of the physical specifications of WirelessHART and how it compares to the physical specifications of IEEE802.15.4. Following the discussion on IEEE802.15.4 we provide a detailed look into the WirelessHART standard and its design in chapter 3.

2.1 Medium Access

When more than two entities need to share the same physical medium we need to implement a protocol for determining how this medium is accessed. There are many solutions to this challenge, but we can coarsely divide these into two groups. The first group is the circuit mode methods which are used to establish "virtual" circuits in which different entities are allowed to communicate with each other without having to compete for medium access. The other group is the packet mode methods. These methods provide rules for allowing entities to communicate when needed using packets on a shared medium and implements rules that make sure that only two entities communicate at the same time.

2.1.1 Circuit Mode Methods

In this subsection we provide a short introduction to the most common methods of circuit mode medium access which is also called virtual circuit switching.

Frequency Division Multiple Access (FDMA)

In FDMA [33] (Figure 2.1) networks a specific frequency range is divided into sub-ranges where each node on the network communicates with its own dedicated frequency. This differs from a FDD (Frequency Division Duplexing) where different sub frequencies are used for up-link and down-link.

Time Division Multiple Access (TDMA)

In TDMA [26] (Figure 2.2) networks one frequency is used for all adjacent nodes, and each node has a specific point in time where it should listen, and when it can send. This "schedule" is known by all nodes. In TDD (Time Division Duplexing) one can emulate a full duplex transmission channel by having specific send time intervals and receive time intervals between two units.

Code Division Multiple Access (CDMA)

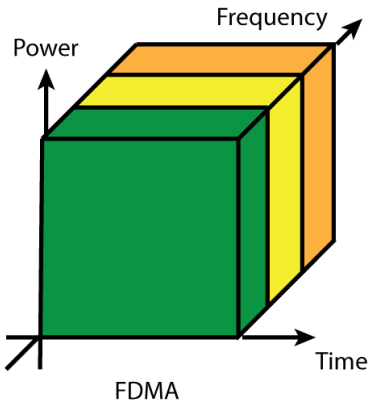


Figure 2.1: Frequency Division Multiple Access - The figure illustrates how the various links are separated by frequency domain while using the time domain simultaneously, which means that multiple nodes may talk at the same time using different frequencies.

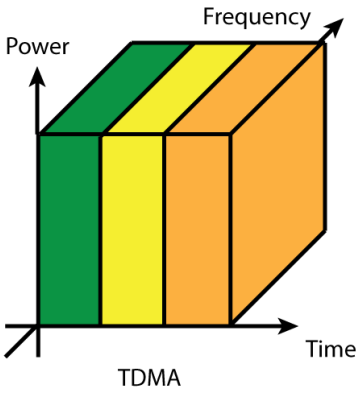


Figure 2.2: Time Division Multiple Access - As displayed by the figure, the time domain is divided into slots while the frequency and power domains remain constant. This essentially means that only one node may utilize the medium at any given time.

In CDMA [45] (Figure 2.3) networks all nodes use a shared medium or a shared frequency but each node has a pseudo-random code that it XOR's its data signal with, hence producing an "encoded" data stream that can only be read by peers having prior knowledge to the senders pseudo-random code.

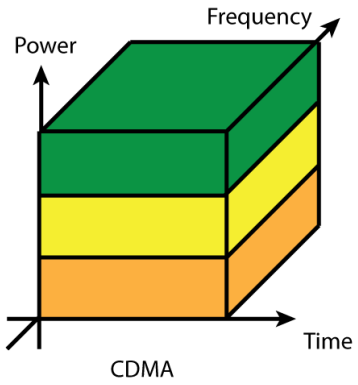


Figure 2.3: Code Division Multiple Access - In CDMA, nodes may utilize the medium at the same time since their individual streams are differently encoded. This is also commonly referred to as stream multiplexing.

2.1.2 Packet Mode Methods

In this subsection we provide a short introduction to the most common methods of packet mode medium access which is also called packet mode switching.

ALOHA

In pure ALOHA [40, 4.2.1],[21] networks, the scheme is simple; if data needs to be sent, it is sent regardless of network states and if a collision takes place, the sender has to try again later.

Slotted ALOHA

In Slotted ALOHA [40, 4.2.1],[21], the improvement to ALOHA is logical time slots where each unit has a time slot for sending and receiving. Thus making it a predecessor to TDMA.

Multiple Access with Collision Avoidance (MACA)

Multiple Access with Collision Avoidance (MACA)[40, 4.2.6] is a scheme that implies that a node that wishes to use a share medium, sends a Request To Send (RTS) where it states the length of what it will send, before the actual sending starts. The receiver will reply

with a Clear To Send message. Nodes hearing either the RTS or the CTS will respectively wait until the CTS is send or wait until the frame has been transmitted.

Carrier Sense Multiple Access with Collision Avoid (CSMA/CA)

CSMA/CA [40, 4.2.2] will use the CDMA protocol, and extend it to avoid collision with the use of the previous mentioned RTS/CTS methodology.

Carrier Sense Multiple Access with Collision Detect (CSMA/CD)

CSMA/CD [40, 4.2.2] will also extend the CDMA protocol, but does not have the functionality to avoid collisions on the network. The sending node will simply check the shared medium before transmissions starts, and if no nodes are sending, it starts its transmission. If a collision is detected after the transmission has started, the node starts a collision recovery algorithm rather than trying to avoid it in the beginning.

The two methods that are most relevant to us is the TDMA and CSMA and these are described in detail in the following sections.

2.1.3 CSMA

Carrier Sense Multiple Access (CSMA) [40, 4.2.2] is a packet mode protocol that helps to make sure only one node at a time transmits on the medium while still providing access to multiple nodes. It was originally designed for collision domain LAN type environments where multiple nodes were using a shared medium to communicate with each other. The protocol works in a "first come, first served" manner. "Carrier Sense" implies that a node is to listen to the medium before talking in order to make sure nobody else is talking i.e. sensing the carrier.

The main disadvantage of CSMA is in the event of a collision; other nodes will not be able to use the medium until all the corrupted frames

finish transmitting. This problem is to a great extent eliminated with the introduction of "Collision Detect" (CSMA/CD) where once a collision is detected a jamming signal is transmitted and both nodes enter into back-off mode for a random period of time.

When moving towards wireless networks this protocol introduces several problems. One of these problems is that it is not possible to send and receive data at the same time with the same radio, so once we start sending it will not be possible to detect collisions.

In order to solve this challenges, going even further and introducing "Collision Avoid" will cause the transmitting node to listen to the medium for a while before it starts to transmit in order to make sure nobody else is talking. If the channel is busy it will wait for a random period of time before transmitting, which greatly decreases the probability of collisions.

2.1.4 TDMA

As with CSMA, Time Division Multiple Access (TDMA) is also a protocol to control access to a shared medium between multiple nodes.

The main idea behind TDMA is to divide the communication channel into slots (Figure 2.4) with a fixed length of time and then assign pairs of nodes (links) to these slots. This allows complete control over which nodes get to send data and when. This requires the slot assignment to be centrally managed and then distributed to all the nodes. One of the main differences to CSMA is that TDMA is deterministic if we disregard the concept of shared slots.

TDMA can be considered a fixed set of resources which can either reduce response time or increase bandwidth based on the slot configuration, bigger slots means higher latency and vice versa. The protocol is optimized for non-continuous burst traffic and has very poor properties when it comes to real-time traffic.

Another challenge when operating in a TDMA environment is the strict requirements in terms of timing. Nodes will not be able to communicate unless they have the same definition of the current time. Although keeping

the time might seem like a simple task, in micro controllers and computer networks this quickly becomes a more complex task.

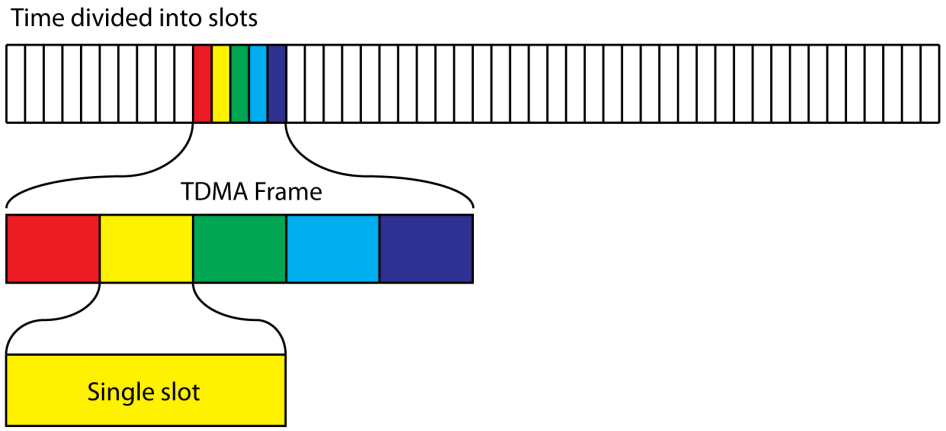


Figure 2.4: The time space is divided into fixed-size slots, each color represents a different node that is allowed to access the medium. The TDMA frame is a fixed sequence of slots that repeats itself and consists of multiple slots. In this figure each frame is divided into 5 slots.

2.1.5 Comparison

Looking at a brief comparison between these two protocols we argue that CSMA should be used when you cannot control access to the medium through some shared management entity. This also means that less interoperability and complexity of management is required.

By extension this implies that the deployment of CSMA environments is cheaper. TDMA is a good alternative if we want to have a deterministic network where all nodes are guaranteed a slot to communicate in, but it also introduces requirements especially when it comes to timing. TDMA is in the WirelessHART standard implemented with a centrally managed entity that manages the superframes and time slots. Ad-hoc scenarios with TDMA are also possible, but this significantly increases the management

overhead as all the nodes in the network need to have the same understanding of how the network topology is structured and which nodes get to talk at what time.

2.2 Standards and Protocols

There are several standards and protocols in wireless networks. In this section we will provide an overview on the most relevant ones for this project.

2.2.1 ISO/OSI Model

The Open Systems Interconnection (OSI) model[30] (Figure 2.5) is a method of thinking about network architecture in terms of abstraction layers. It is considered the de-facto model of which all layered models are compared against and serves as a good guide for designing layered and extendible systems.

The OSI model defines 7 layers which each provide their own separate area of responsibility. Each of the layers are dependent on the functionality of the layer beneath it in the stack. In other words, each layer serves the layer above it in the stack, and each layer is served by the layer below it in the stack.

The layers in the OSI model displayed in figure 2.5 in top to bottom order are:

Layer 7 - Application

This is the level that the user interacts with, and where the user-space applications work and communicate with the rest of the network stack.

Layer 6 - Presentation

The presentation layer is responsible for translating programming languages and machine-readable content into human-readable con-

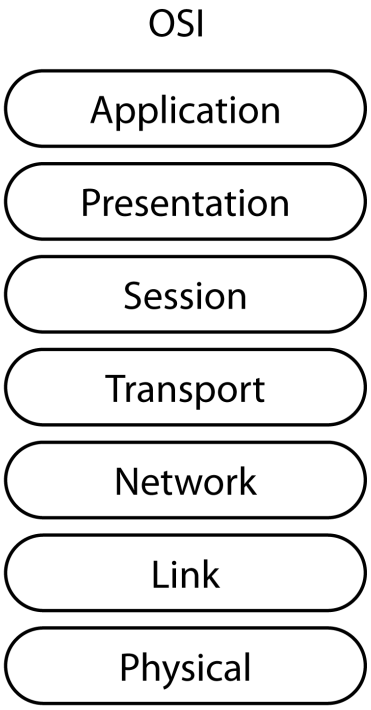


Figure 2.5: OSI Model - Basic layer overview

tent. This layer is usually hard to separate from the application layer.

Layer 5 - Session

The session layer is responsible for maintaining sessions between hosts, sessions can be thought of as private conversations between two entities.

Layer 4 - Transport

The responsibility of the transport layer is to ensure that data is transported error-free to its destination and provide error-recovery methods should this fail for some reason.

Layer 3 - Network

The network layer is responsible for making the decision of which path the data should follow in order to reach its final destination.

Layer 2 - Data Link

Layer two provides a system through which network devices can share the communication channel. A more detailed look at the various available medium access protocols are defined in section 2.1.

Layer 1 - Physical

This layer defines the electrical and physical characteristics of the communication device and transmission medium and the interface between the medium and the device.

2.2.2 IEEE 802.11 - Wireless Local Area Network (WLAN)

The purpose of the IEEE 802.11 standard[5] is to define several physical layers and one medium access control layer for wireless connectivity between stationary and mobile nodes. It supports infrastructure and ad-hoc modes as compared to IEEE 802.15.4 which only supports ad-hoc networks. IEEE 802.11 operates in the 2.4GHz ISM band and divides this band into 13 channels (Figure 2.6). There is also an additional 14th channel that was introduced in Japan, and is only allowed in 802.11b. But for this project we will ignore channels that are not generally available.

There are several versions of the 802.11 protocol. These version include 802.11a, 802.11b, 802.11g and recently also 802.11n. 802.11g is the most widespread in use today so we will use this for the comparison. The bandwidth is limited to 54Mbps and the range is somewhere between 50-100m in a normal operating environment. 802.11 uses CSMA/CA for medium access as compared to its wired equivalent 802.3 (Ethernet) which uses CSMA/CD. One of the primary limitations of 802.11 when it comes to operation compared to 802.15.4, is the high power consumption.

2.2.3 IEEE 802.15.4 - Low Rate Wireless Personal Area Network (LR-WPAN)

The IEEE 802.15 TG4[15] is intended for low data-rate nodes with battery lifetime ranging from multi-month to multi-year. It supports short (16-bit)

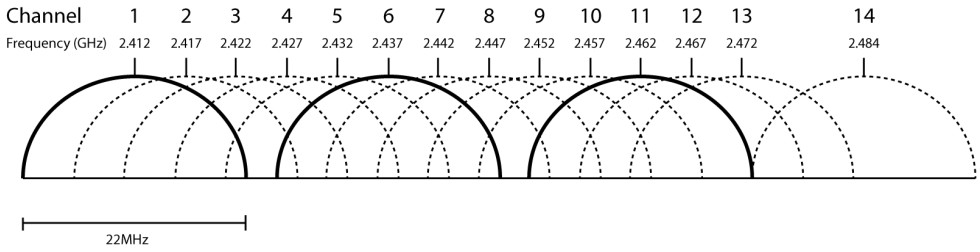


Figure 2.6: IEEE 802.11 Channels - The figure provides an overview over the available channels in the 2.4GHz ISM band along with their center frequencies and channel widths. There are only 3 non-overlapping channels available in this band.

and standard 64-Bit IEEE addressing modes in an automatically coordinated network. It operates in the 2.4GHz ISM band as well as in the 915 MHz and 868 MHz band. Compared to IEEE 802.11, IEEE 802.15.4 has a lower data-rate which in turn naturally also results in a much lower power consumption. In short, compared to 802.11 one is trading high bandwidth for battery life and reliable communication. IEEE 802.15.4 splits the ISM band into 16 channels which range from 11-26 (Figure 2.7), however, we ignore channel 26 as it is not allowed in some regions.

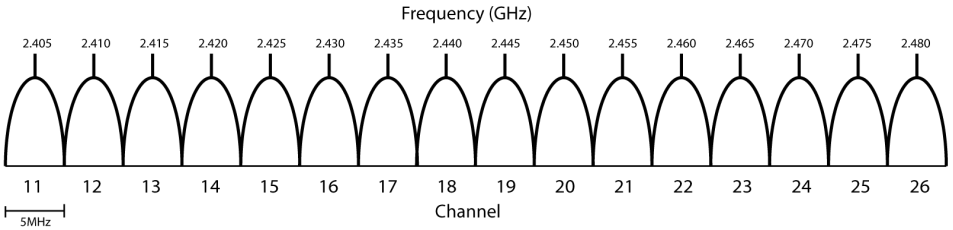


Figure 2.7: IEEE 802.15.4 Channels - The figure provides an overview over the available channels in the 2.4GHz ISM band along with their center frequencies. There are 16 available non-overlapping channels which are 5MHz wide each.

2.2.3.1 Microchip Wireless (MiWi) and MiWi P2P

MiWi and MiWi P2P [9] are proprietary protocols which are designed by Microchip Technology. They are based on the IEEE 802.15.4 standard and their usage area is in low-data-rate and short-range wireless networks. The standard mentions industrial monitoring, home automation and wireless sensors as potential use-areas.

2.2.3.2 ZigBee

ZigBee[19] is a wireless standard based on IEEE 802.15.4 intended for use in Low-Rate Wireless Personal Area Networks (LR-WPANs). This in combination with low-cost radio frequency (RF) modules enables widespread usage in for example wireless light switches, remote controls, in-home meter displays and other consumer electronics.

2.2.3.3 WirelessHART

As previously mentioned WirelessHART is a wireless version of HART with the physical layer being based on IEEE 802.15.4 using the 2.4GHz ISM band. The main goal is to provide reliable, short-range and time-based access to communication channels. In WirelessHART we are able to make certain assumptions and impose some statically configured parameters in the code in order to make the implementation simpler. These areas will be pointed out in chapter 5. WirelessHART uses the TDMA medium access protocol to control the communication medium. The design and operation of WirelessHART in comparison to IEEE 802.15.4 will be further discussed in sections 2.3 and 3.

2.2.4 ISA100.11a

ISA100.11a[16] is an open wireless networking technology standard developed by the International Society of Automation (ISA). This protocol is also specified in a manner that allows it to communicate in the same 2.4GHz ISM band as WirelessHART using IEEE 802.15.4 on the physical

layer. In addition ISA100.11a supports the functionality of being a data carrier for different protocols and thus is not limited only to HART, but also supports other protocols such as Fieldbus[32] and Profibus[25].

2.2.5 ROLL

The Internet Engineering Task Force (IETF) has an active working group working on the issues of Low power and Lossy Networks (LLN), more specifically Routing Over LLN (ROLL[13]). The ROLL working group's specific example of a LLN is a wireless sensor network making ROLL very interesting for routing in our WirelessHART network. This working group, which was created in February of 2008, has been looking at the issues with LLN and working on a routing protocol for LLNs which is called the Routing Protocol for LLN (RPL).

2.3 IEEE 802.15.4

The IEEE 802.15 working group has the primary goal to maintain and specify Wireless Personal Area Network (WPAN) standards. Other task groups within the same family as for example WPAN/Bluetooth and High Rate WPAN (HR-WPAN). IEEE 802.15.4 is a standard for the specification of low-rate wireless personal area networks (LR-WPANs). The primary goal of the LR-WPANs are to provide low complexity and very long battery life. The standard defines the first two layers of the OSI model namely the physical layer and the link layer.

2.3.1 Components

IEEE 802.15.4 defines two primary types of network nodes, full-function devices (FFD) and reduced-function devices (RFD). The FFD device has all the functionality required to serve as a network coordinator while the RFD devices are extremely simple and sometimes very limited on resources and functionality. RFD devices may not act as network coordinators.

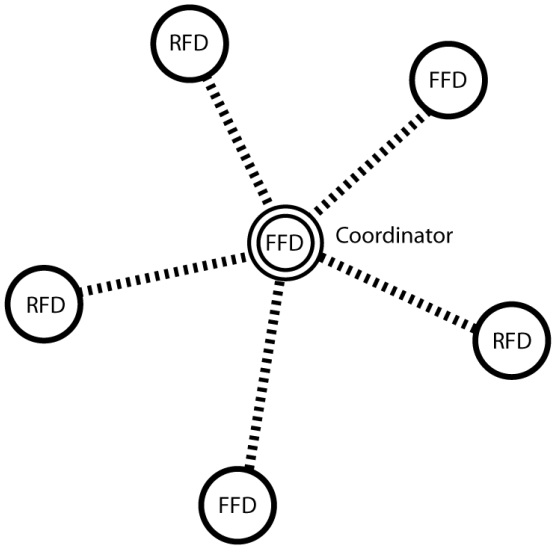


Figure 2.8: Star topology - The PAN coordinator is represented by the double lined node and is responsible for managing the network, all traffic goes through the coordinator. The network is limited to single hop.

2.3.2 Topologies

There are two basic topologies for IEEE 802.15.4, these are the star(Figure 2.8) and peer-to-peer(Figure 2.9) topologies. The simplest topology is the star topology and it consists of a coordinator surrounded by other devices and all traffic flows through the coordinator.

The peer-to-peer topology is a bit more advanced and allows devices to communicate with each other directly without going through the coordinator. This requires that any nodes that are not leaf nodes need to be full function devices in order to be capable of routing messages.

2.3.3 Service Primitives

Interactions between the layers in a service model are defined by service primitives where primitives primarily means operations. IEEE 802.15.4 defines four types of generic service primitives. The basic operation of these service primitives are display in figure 2.10.

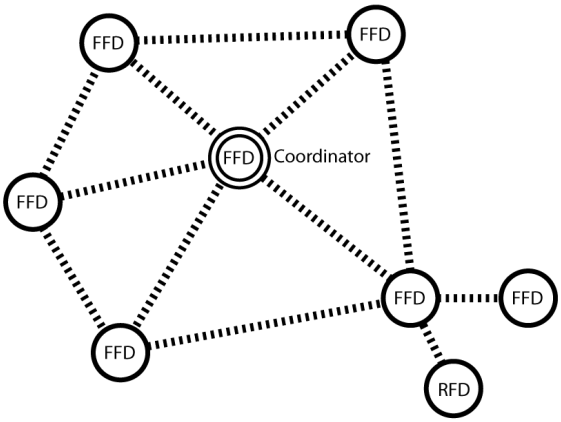


Figure 2.9: Peer to peer (Mesh) topology - The PAN coordinator is still responsible for managing the network but the nodes are also allowed to communicate directly with each other. In addition, a mesh topology opens the possibility for multi-hop networks.

REQUEST

This service primitive is used utilized by the upper layer to ask the layer further down in the protocol stack to provide a service.

CONFIRM

This service primitive is utilized by the lower layer to mitigate the result of a previous request back to the upper layer.

INDICATION

This service primitive is utilized by the lower layer to notify the upper layer about an event.

RESPONSE

This service primitive is utilized by the lower layer to conclude a previously triggered `INDICATION`.

2.3.4 Physical Layer (PHY)

The physical layer in IEEE 802.15.4 is responsible for providing access to the data transmission services to the link layer. The PHY layer manages

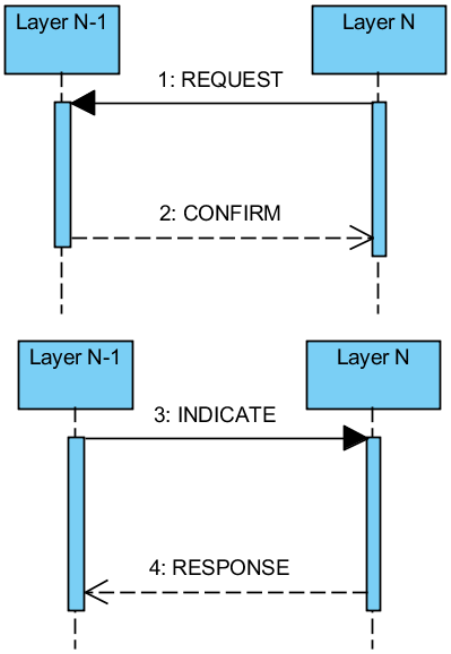


Figure 2.10: Types of service primitives and the interactions between them, as we can see `REQUEST` is directed towards a lower layers and is acknowledged by a `CONFIRM`, and `INDICATE` is directed to a higher layer and is acknowledged by a `RESPONSE`.

the radio and the interface to access the radio. The specification defines 3 frequencies (Defined in table 2.1). The 2400-2483.5MHz range overlaps with the industrial, scientific and medical (ISM) band defined by ITU-R[43].

Frequency range (MHz)	**Bit rate**	**Number of channels**
868-868.6 MHz	20, 100, 250 kb/s	1
902-928 MHz	40, 250 kb/s	30
2400-2483.5 MHz	250 kb/s	16

Table 2.1: Frequency ranges in IEEE 802.15.4-2006

2.3.5 Medium Access Control Layer (MAC)

The IEEE 802.15.4 media access control layer (MAC) is responsible for providing access to a shared physical channel. The MAC layer in IEEE 802.15.4 also provides reliable data transfers and collision avoidance through its utilization of the previously discussed CSMA/CA. It has functionality for providing frame validation, beaconing and the management of time slots. The standard defines a superframe structure (Figure 2.11) where superframes are separated by beacons. Each superframe consists of 16 slots where communication between the nodes may occur after which there is an inactive period where the nodes may sleep in order to conserve power. These slots are shared between all the nodes so nodes that wish to send data need to contend for these slots using CSMA/CA. The Coordinator may also allocate some of these slots statically to nodes, allowing contention-free access to the physical medium.

2.3.5.1 IEEE 802.15.4 MAC frame format

The IEEE 802.15.4 standard defines several types of frames, these include data frames, acknowledgment frames, command frames and beacon frames. As all the frames consist of many of the same header fields we only discuss the data frame format in this section. The data frame format is what is relevant to the MAC layer used by WirelessHART.

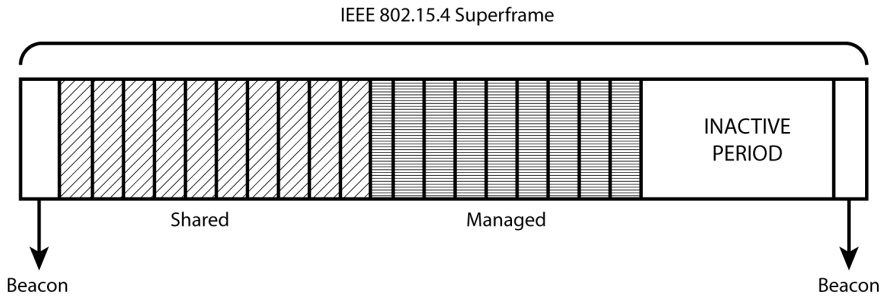


Figure 2.11: IEEE 802.15.4 superframe - The superframes are separated by beacons and consist of a series of shared slots that may be contended for using CSMA/CA, in addition there may be several allocated slots which are contention-free. The frame also consists of an inactive period where the radio may sleep in order to conserve power.

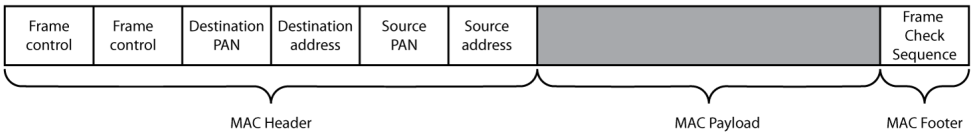


Figure 2.12: IEEE 802.15.4 MAC header/frame

As displayed in figure 2.12 the standard IEEE 802.15.4 frame consists of 3 main parts, namely the MAC header, the MAC payload and the MAC footer. The header is what is most interesting to us and the MAC header consists of the following fields.

- Frame Control
- Sequence Number
- Destination PAN
- Destination Address
- Source PAN
- Source Address

The MAC frame header of the IEEE 802.15.4 layer overlaps with the frame header format utilized by WirelessHART. Since this is the case, tools for analyzing and decoding IEEE 802.15.4 traffic may be used also on WirelessHART traffic. The differences between these two frame formats and how they overlap each other is further described in section 3.3.5.

2.4 Chapter Summary

Wireless sensor networks is a complicated subject and many standards and protocols which share the same physical layer specifications do not bear the slightest resemblance in the higher layers. Throughout this chapter we have provided a brief introduction to the concepts and challenges of medium access and a comparison between the different methods that exist. We have also briefly looked at the popular IEEE 802.11 standard and compared it to the IEEE 802.15.4 standard and then looked at some of the most popular protocols built on this physical standard. At the end of the chapter we have provided an introduction to the components of a IEEE 802.15.4 network and the physical specifications of IEEE 802.15.4.

Chapter 3

WirelessHART

In this chapter we provide a detailed explanation on the background of the WirelessHART protocol and how it maps onto the standard 7 layers of the OSI model. Before we can go in detail on how the WirelessHART protocol has been implemented during the course of this project we provide some background on WirelessHART including an in-depth explanation of the most relevant layers of the WirelessHART protocol (Figure 3.1). This includes the physical layer (PHY), the link layer (DLL) and the network layer (NL). The transport layer and higher layers are handled by HART and are not relevant to this project, thus they will not be discussed in detail. In addition to a detailed look into the layers of the WirelessHART specification we will also provide an overview of the types of network components commonly found in a WirelessHART network. At the end of the chapter we also provide an explanation of the various challenges and requirements in terms of time synchronization and a walk-through of the WirelessHART join process.

3.1 Background

WirelessHART is the successor of the Highway Addressable Remote Transducer protocol (HART) which is an early Fieldbus protocol[32] developed by Rosemount Inc. in the beginning of the 1980s. The protocol has been

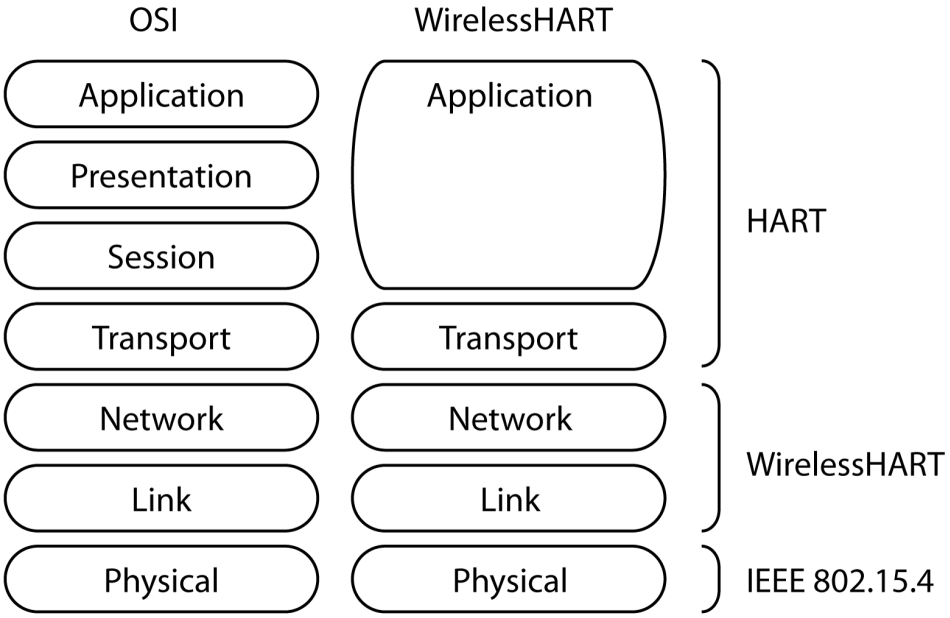


Figure 3.1: OSI Model and its relation to the layered model used to describe the WirelessHART, the physical layer is that of IEEE 802.15.4, WirelessHART defines the link and network layer while the higher layers are handled by the HART protocol.

publically available since it was made an open protocol by Rosemount Inc. in 1986. Currently the HART protocol is controlled by the HART Communication Foundation (HCF) which is a non-profit organization founded in 1993. One of the primary reasons that HART is present in over 26 million devices world wide is the fact that it is designed to communicate over legacy 4-20mA wiring, thus allowing easy transitions from legacy systems. HART is considered a simple and robust protocol and WirelessHART aims to transfer these characteristics into the wireless domain (Figure 3.2).

The WirelessHART protocol utilizes the physical layer of the IEEE 802.15.4 standard in addition to defining its own link and network layer. The layers above the network layer, namely the transport and application layers are provided by the original HART standard.

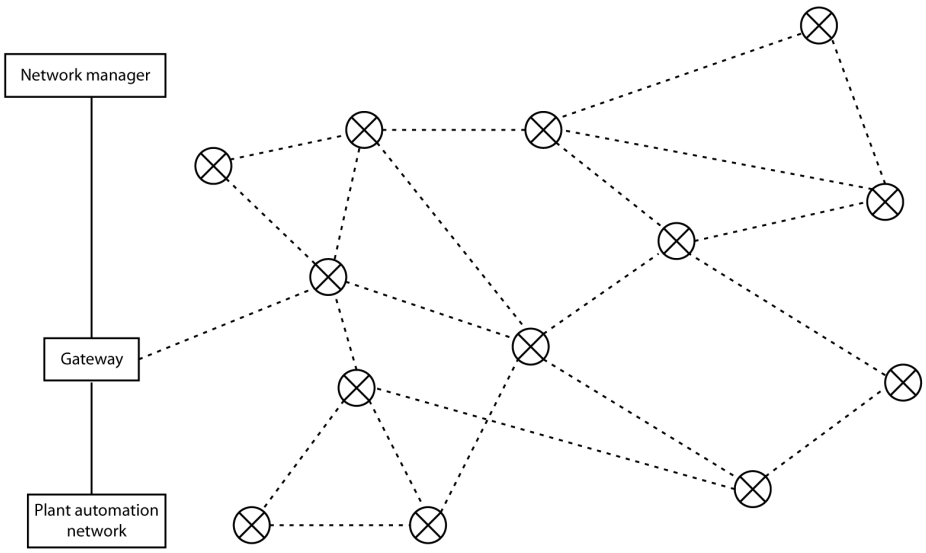


Figure 3.2: An example WirelessHART network showing a multi-hop mesh connected to a WirelessHART Gateway. The Gateway is in turn connected to a WirelessHART Network Manager and the plant automation network (production backbone), which in turn can be any combination of wired or wireless technology.

3.2 Physical Layer

The WirelessHART protocol utilizes the physical layer in IEEE 802.15.4. In this section we provide an in-depth overview of the physical characteristics of IEEE 802.15.4 from a general perspective.

3.2.1 Frequency Range

While the IEEE 802.15.4 standard defines several methods of modulation and frequency ranges available for use, WirelessHART only supports the 2.4GHz frequency range (2400-2483.5 MHz) using O-QPSK (Offset quadrature phase-shift keying) [37] modulation and DSSS (Direct-sequence spread spectrum). The communication rate is defined in kchip/s which describes how many pulses of DSSS code is transmitted per second. The full set of physical requirements for WirelessHART are defined in table 3.1.

Technology	**Requirement**
Transmission technique	DSSS
Frequency range	2400-2483.5 MHz
Communication rate	2000 kchip/s
Modulation	O-QPSK
Bit rate	250 kb/s
Symbol rate	62.5 ksymbol/s
Symbols	16-ary orthogonal

Table 3.1: Physical requirements in WirelessHART

3.2.2 Channels

The frequency range is divided into multiple channels as described in section 2.2.3, in the case of IEEE 802.15.4 there are 16 channels (numbered from 11-26), however, channel 26 is not allowed in all regions. Each of the channels cover a frequency range of 5MHz. An overview of the available channels can be seen in figure 2.7.

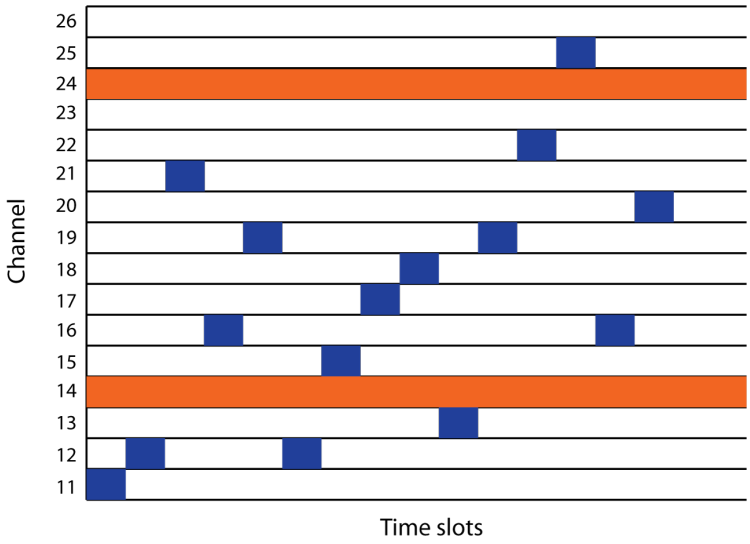


Figure 3.3: Channel hopping is when the channel used for transmission is pseudo-randomly selected using a selection algorithm known to both sides of the conversation. The frequencies marked as orange are channels that are blacklisted and thus ignored in the selection process. Blue represents slots that are used.

3.2.3 Clear Channel Assessment

Clear Channel Assessment (CCA) is the process of checking that the channel is not used (clear) before transmission. There are three primary types of CCA. WirelessHART uses CCA mode 2 which is carrier sense CCA. Mode 2 will consider the medium busy if a signal which is detected as another IEEE 802.15.4 transmission is detected.

3.2.4 Channel Hopping

Channel hopping (also called frequency hopping) is a technique utilized to avoid interference and increase randomness during transmission to improve security and reliability. A pseudo-random sequence is used by the nodes to determine which channel to use so that only the authenticated nodes may know which channel will be used at any given time. In addition this tech-

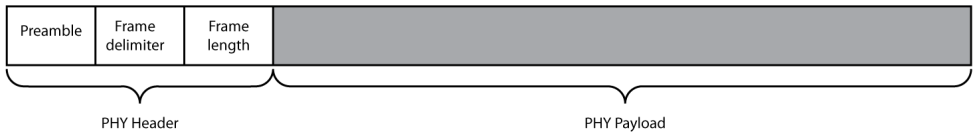


Figure 3.4: Physical Protocol Data Unit (PPDU)

nique reduces the impact of interference experienced on certain channels allowing communication to take place on other channels. Channels may also be blacklisted and thus they will not be used in the hopping sequence. A visual representation of channel hopping is shown in figure 3.3.

3.2.5 Physical Layer Primary Data Unit

The physical layer primary data unit (PPDU) as displayed in figure 3.4 consists of a preamble, a delimiter, a length field and the data link layer payload.

3.2.6 Service Primitives

The service primitives of the physical layer are defined by the WirelessHART standard. The standard does not specify how these services primitives should be implemented. We separate the primitives into 5 groups, `ENABLE`, `CCA`, `DATA`, `ERROR` and `LOCAL_MANAGEMENT`. As described in section 2.3.3 requests are utilized by the higher layer to communicate with the lower layer, confirm is used as a response to a request and indicate is used for the lower layer to initiate contact with the higher layer.

ENABLE.request

This service primitive is used by the MAC layer to change the state of the radio to either receive (RX) or transmit (TX) mode.

ENABLE.confirm

This service primitive is used to respond to a `ENABLE.request` with a status code that informs the caller if the request was a success.

ENABLE.indicate

This service primitive is used by the physical layer to notify that MAC layer that data is coming in on the radio.

CCA.request

This service primitive is used by the MAC layer to initiate Clear-Channel-Assessment to determine if the communication channel is free.

CCA.confirm

Response to the `CCA.request` with the error condition should the request not have been completed.

DATA.request

This service primitive is used by the MAC layer to ask the physical layer to transmit a packet.

DATA.confirm

This service primitive is used by the physical layer to notify the MAC layer whether or not a transmission was completed without errors.

DATA.indicate

This service primitive opposed to `ENABLE.indicate` is used to notify the MAC layer that a whole packet has arrived on the radio and is ready to be read.

ERROR.indicate

This service primitive is used by the physical layer to notify the MAC layer that the reception of a packet failed.

In addition the standard defines a set of `LOCAL_MANAGEMENT` primitives, these are used for performing configuration of the physical layer and are structured the same way as the other primitives namely with a request, confirm and indicate function.

3.3 Data Link Layer

In this section we provide a detailed description of the WirelessHART Data Link Layer including the TDMA State Machine, methods of Time Synchronization and how the WirelessHART Data Link Layer is subdivided into the MAC and LLC sub-layers. In addition we provide an introduction to the link scheduler in WirelessHART and the definition of Service Primitives (SP) on the Data Link Layer.

3.3.1 Time Division Multiple Access

As previously discussed, WirelessHART uses TDMA in combination with channel hopping to schedule physical media access to links. In order to manage this it uses superframes (Figure 3.5), which are multiple time slots aligned after one another in a fixed sequence. Superframes are time-limited to a fixed number of slots, where each slot lasts for 10ms. The superframes are repeated continuously and may be active or inactive. WirelessHART requires that there is always at least one superframe active in the network. In each slot two nodes are allowed to communicate with each other, there are no limitations on whether or not these two nodes are scheduled multiple slots during a superframe. In reality, this can be thought of as a virtual link defined by a slot in a superframe, and a channel offset. A channel offset is a relative value indicating which channel to use, the value is to be added or subtracted from the current channel.

3.3.2 TDMA State Machine

The primary purpose of a state machine is to clearly defined the possible states in the system and the transitions that exist between them. The state machine in the WirelessHART link layer is clearly defined and consists of 6 different states.

If we start by looking at figure 3.6 we can see that the default fallback state of the state machine is `IDLE`. In Tegelsrud and Frøysadal's implementation only the `IDLE` state is actively used. This means that in

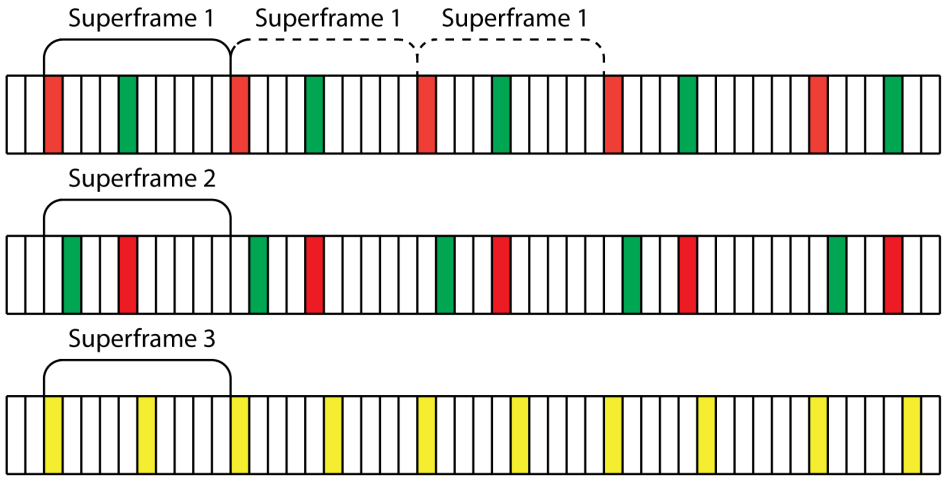


Figure 3.5: Superframes in WirelessHART - The figure displays 3 different superframes, each with their own sequence of links. Each of the superframe repeat themselves over time and whenever there are more than one link to be scheduled at the same time the link scheduler is responsible for the selection process. As as example, yellow slots in superframe 3 may be allocated for management traffic and highest priority, making these take precedence over superframes 1 and 2.

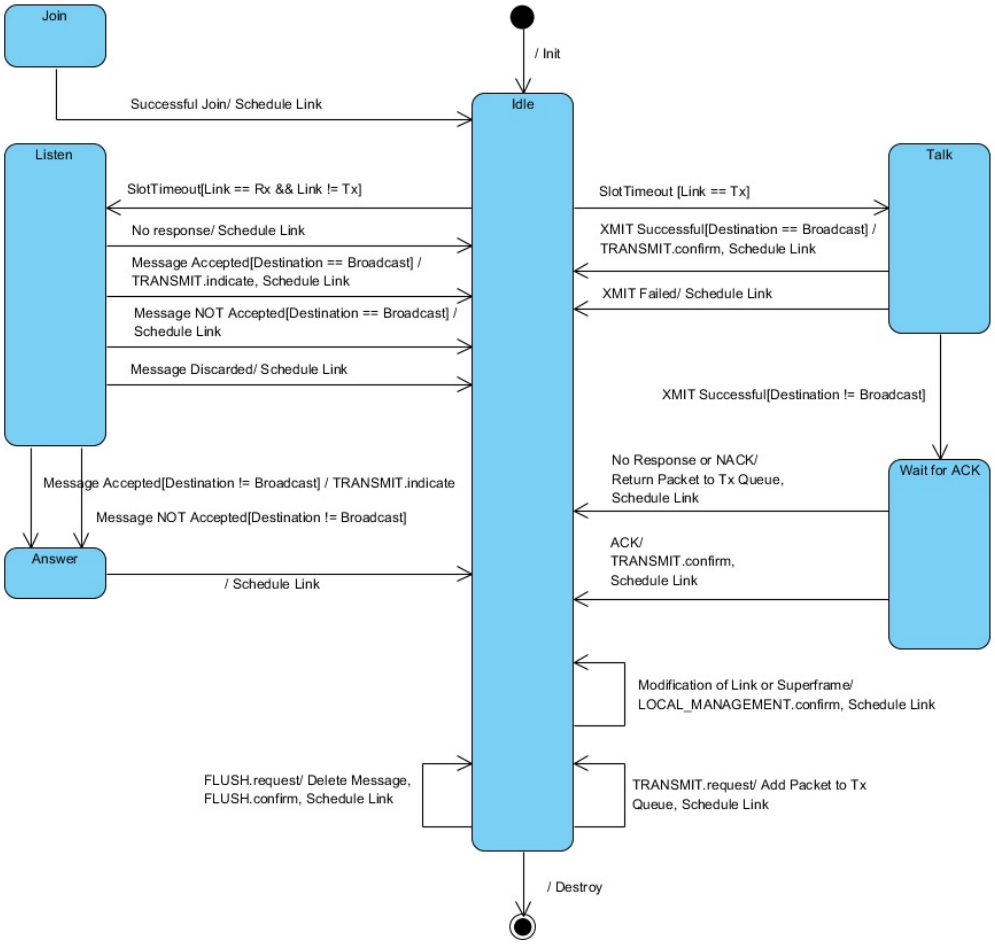


Figure 3.6: TDMA State Machine

order to fully implement a state machine we need to design and implement the states and the transitions between them.

JOIN

The `JOIN` state can be further separated into two sub-states, active join and passive join. The protocol dictates that we should first enter an active join state for a pre-configured amount of time. During the active join the radio will continuously listen for incoming packets and use these to synchronize and join with the network. If no packets have been received during active join, the node will transition into passive join state where it will wake up from time to time to listen for

packets and then go back to sleep. The JOIN state is the first states any node will enter and is absolutely critical for the implementation of a properly synchronized network with more than one node.

IDLE

Once the node is in IDLE state a set of condition will be required for anything to happen. The two most important triggers for exiting the IDLE state is the task of servicing a TX or RX slot, thus transitioning into the TALK or LISTEN state respectively. Other events that may happen while in the IDLE state is the modification of a device's list of superframes or links or a flush request causing a packet to be discarded.

TALK

The TALK state is entered once the device is ready to transmit a packet and only returns to IDLE when a packet is successfully sent or a failure occurred. Should the packet require an ACK the state will transition directly to the WAIT state without entering IDLE state. Once the TALK state is entered the XMIT engine takes control and the XMIT engine will be more thoroughly described in section 6.5.3.

LISTEN

The LISTEN state is entered from the IDLE state when the node is ready to serve an RX slot, the behavior of the LISTEN state is more thoroughly described in section 6.5.4.

WAIT

The WAIT state (Wait for ACK) is entered after a packet which requires an acknowledgment has been successfully transmitted and will always transition back into IDLE state.

ANSWER

The ANSWER state is entered when an incoming packet requires an ACK to be sent and will always transition back into IDLE state.

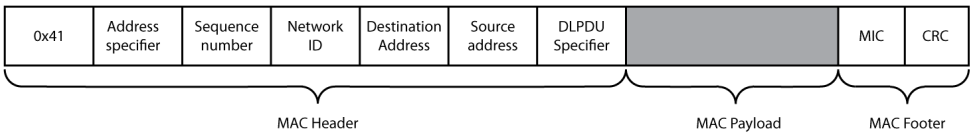


Figure 3.7: Data Link Protocol Data Unit (DLPDU)

3.3.3 Time Synchronization

In WirelessHART, the length of a slot in a superframe is 10ms. This means that time slots need a high level of precision in timing. The WirelessHART standard provides some strict timing requirements that need to be respected in order to communication to occur reliably. The timing requirements operate with a maximum uncertainty of 100μs and will be further discussed in chapter 6.

3.3.4 Layer Subdivision

The link layer can be further subdivided into two layers, namely the Logical Link Control (LLC) and the Medium Access Control (MAC) layer. The LLC layer is responsible for managing the links, data structures and scheduling while the MAC layer is responsible for controlling access to the communication medium.

3.3.5 Data Link Layer Protocol Data Unit

The Data Link Layer Protocol Data Unit (DLPDU) is the link layer data structure that is used to encapsulate network layer traffic before it is sent to the physical layer for transmission. Figure 3.7 shows the DLPDU which contains several fields for defining the link layer properties of the packet. An overview of these properties are listed in table 3.2. These headers also overlap to some degree with the IEEE 802.15.4 frame format header described in section 2.3.5.1.

0x41

Position	Name/Description	Length in Bytes
1	0x41	1
2	Address Specifier	1
3	Sequence Number	1
4	Network ID	2
5	Destination	2 or 8
6	Source	2 or 8
7	DLPDU Specifier	1
8	Payload	Depends on DLPDU type
9	Message Integrity Code (MIC)	4
10	Cyclic Redundancy Check (CRC)	2

Table 3.2: The DLPDU Components

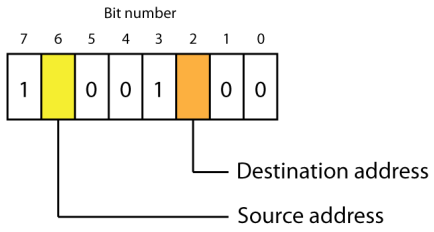


Figure 3.8: The DLPDUs Address Specifier byte

The standard defines that the first byte in the DLPDU should always be set to the hexadecimal value `0x41`.

Address Specifier

The address specifier is a byte which specifies whether the long address mode is used on the source and/or destination address. As figure 3.8 shows bits 7 and 3 are always set and if bit 6 is set the source address is 8 bytes and if bit 2 is set the destination address will be 8 bytes. If they are not set the device nickname will be used, which is 2 bytes long.

Sequence Number

The sequence number is the least significant byte of the Absolute Slot Number (ASN) at the time of transmission.

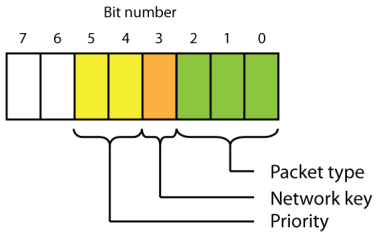


Figure 3.9: The DLPDU Specifier byte

Network ID

The network ID is the name of the network the packet is meant for. If the network ID does not match the field device's network ID the packet is ignored.

Source and Destination Address

Depending on whether or not the address specifier bits have been set or not, the destination and source addresses are either 2 or 8 bytes long. The fields indicate the neighbor sending the packet and which device is the destination. The original source and final destination are specified in the Network Layer PDU, more on this in section 3.4.1.

DLPDU Specifier

The DLPDU specifier is a one byte field which specifies the priority, network key and the type of the DLPDU. Figure 3.9 shows the bits of the specifier; bits 7 and 6 are reserved and set to 0, bits 5 and 4 indicate the priority of the DLPDU, bit 3 is the network key (used for authentication of the DLPDU) and the last three bits define the packet type. The different types are further described in section 3.3.5.1.

Payload

The DLPDU payload depends on the packet type described in the DLPDU specifier, these will also be described further in the next section.

Keyed Message Integrity Code (MIC)

The MIC ensures the integrity of the DLPDU and is calculated using the network key. If the MIC calculated by the receiver does not match the one in the DLPDU the packet will be discarded.

Cyclic Redundancy Check

This field is calculated in hardware and is specified by the IEEE 802.15.4 standard[20].

3.3.5.1 DLPDU Packet Types

Below is a description of the different types of DLPDU packets defined in the WirelessHART standard.

Data DLPDU

A Data DLPDU indicates that the packet received contains data that is to be sent to a final destination set in the NPDU.

ACK DLPDU

An ACK DLPDU is sent directly from one neighbor to another and used to acknowledge the receiving of any DLPDU, except broadcast and other ACK DLPDUs. It consists of a response code which is 0 if the transmission was successful and 61, 62 or 63 if an error occurred. It also contains a 2 byte time adjustment in microseconds which indicates how much (negative number) earlier or (positive number) later than expected the packet being acknowledged arrived.

Advertise DLPDU

The Advertise DLPDU is used to advertise the network, passing with it enough information to allow new nodes to synchronize themselves in a way that allows them to initiate a join sequence. The Advertise DLPDU contains basic information such as the current ASN, a channel map of allowed channels and a list of join links that can be used to initiate the join sequence.

Keep-Alive DLPDU

This DLPDU has three main purposes: time synchronization, con-

firming connectivity and neighbor discovery. A keep-alive DLPDU is sent and synchronization is based on the time adjustment received in the ACK DLPDU of the neighbor. It is also sent to neighbors to inform them that the sender is still running and a device can be instructed to send out keep-alive DLPDUs in order to make itself known to new neighbors.

Disconnect DLPDU

When a field device is leaving the network it sends out a disconnect DLPDU. A device receiving such a DLPDU will delete the node from its neighbors list and all links to the device must be deleted as well.

3.3.6 Link Scheduler

The task of the link scheduler is to be able to, at any time, select the next link that should be processed. While scheduling may seem like a straightforward task initially the reality is quite different as there are many factors in play. These factors include multiple superframes with different priorities and links which also have different priorities.

3.3.7 Service Primitives

The service primitives of the MAC layer are defined by the WirelessHART standard. These services primitives are separated into transmit, network events, receive and local management. The `TRANSMIT` service primitive is the only primitive that is required by the standard, the rest are optional.

TRANSMIT.request

This service primitive is used by the network layer to transmit data to another device, after the link layer has created a packet and it is ready to send it will call `TRANSMIT.request` which will ask the MAC layer to transmit the packet.

TRANSMIT.confirm

This service primitive is a response to a `TRANSMIT.request` and is used

to notify the network layer about the result of a previous request.

TRANSMIT.indicate

This service primitive is used by the MAC layer to notify the network layer about successful reception of a packet that is addressed to this device.

FLUSH.request

This service primitive takes a packet handle as an argument and is used to delete the packet referenced by the handle.

FLUSH.confirm

This service primitive is used as a response to a previous request.

DISCONNECT.indicate

This service primitive is used to notify the network layer that a device is leaving the network, allowing the MAC layer to make the necessary adjustments.

PATH_FAILURE.indicate

This service primitive is used to notify the network layer that the path to another device has failed.

ADVERTISE.indicate

This service primitive is used to notify the network layer about the reception of an advertisement, the network layer can then use this information in order to successfully synchronize with the network.

NEIGHBOR.indicate

This service primitive is used to notify the network layer about the reception of a packet from a device which is not listed in the neighbor table allowing the network layer to make the necessary adjustments.

RECEIVE.indicate

This service primitive is used to indicate that a packet which was not addressed to the device was received.

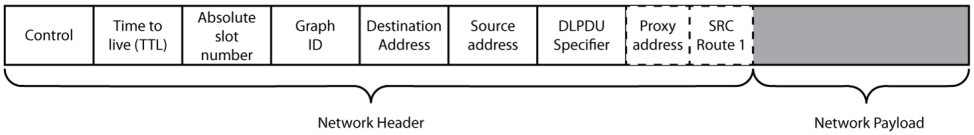


Figure 3.10: Network Layer Protocol Data Unit (NPDU)

In addition the standard defines a set of `LOCAL_MANAGEMENT` primitives[29, 5.3.3], these are used for performing configuration of the MAC layer and are structured the same way as the other primitives namely with a request, confirm and indicate function.

3.4 Network Layer

The Network Layers primary tasks are routing, end-to-end security and transport services. This section gives a more detailed description of the WirelessHART Network Layer.

3.4.1 Network Layer Protocol Data Unit

Figure 3.10 shows the Network Layer Protocol Data Unit (NPDU) structure specified by the standard. Table 3.3 lists the fields and how many bytes they each occupy, each of these fields is described in more detail later in this section.

Position	**Name/Description**	**Length in Bytes**
1	Control Field	1
2	Time To Live (TTL)	1
3	Part of the Absolute Slot Number (ASN)	2
4	Graph ID	2
5	Final Destination	2 or 8
6	Original Source	2 or 8
7-9	Optional Routing Fields	0-18

Table 3.3: The NPDU Components

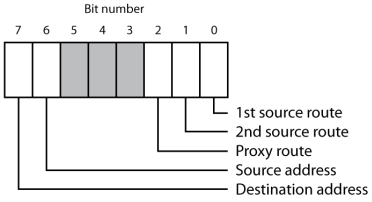


Figure 3.11: The NPDU Control Byte

Control Field

This is a one byte (8 bits) field used to describe some features of the NPDU. Figure 3.11 shows the Control Byte and what the different bits indicate when set to 1. Bits 7 and 6, when set, indicate that long address mode is used (8 bytes) instead of nicknames (2 bytes) on destination and source addresses respectively. The next three bits, 5-3, are reserved for future use and set to 0. The last three are for the optional routing fields the NPDU can contain. When bit 2 is set a Proxy Route is used and bits 1-0 indicate weather or not Source Routes are used, further information on these is provided later in this section.

TTL

The Time To Live (TTL) field is a 1 byte value that defines the packet lifetime. If the device receiving the packet is not the final destination the TTL value is decremented by one, unless the TTL is 0 in which case it has expired and the packet is not to be forwarded. A TTL set to 0xFF shall not be decremented and will be forwarded until it reaches its destination.

ASN

The 2 byte Absolute Slot Number (ASN) field contains the 16 least significant bits of the network ASN.

Source and Destination

The final destination and original source addresses are found in their respective fields in the NPDU. The original source describes which

device originally sent the packet and the final destination describes which device the packet is intended for. Depending on whether bits 7 and 6 in the Control Field were set or not, the fields can either contain a 2 byte nickname or an 8 byte long address.

Graph ID

The Graph ID points to a list of nodes for routing. The packet can be forwarded to any of these nodes (devices) in order to get the packet to its final destination. If the optional routing fields are present these will be taken into consideration together with the list of nodes from the Graph ID when forwarding a packet.

Proxy Route (optional)

This optional field is used when the final destination is a device that is not a full-worthy member of the WirelessHART network, either because it is still joining the network or because it is in quarantine (More on this in section 3.7). The 2 byte nickname specified in this field points to the device that is responsible for communication with the non-full worthy member of the network. This means that when the packet arrives at the device in the proxy route field this device will forward the packet to its neighbor that is the final destination.

Source Route (optional)

This field should only be used when debugging or testing paths in the network. Up to two of these fields can be added to the NPDU and each contains four 2 byte nicknames of devices in the network. When this/these fields are present the packet is to be routed through these devices until it reaches its final destination.

3.4.2 Service Primitives

Below is a short description of all the Network Layer Service Primitives (SP).

TRANSMIT.request

This service primitive is used to transmit a packet on the network layer.

TRANSMIT.indicate

This service primitive is triggered when a packet is received on the Network Layer and the payload is ready to be read by the higher layer.

TRANSMIT.response

This service primitive is used by the device to respond to any `TRANSMIT .indicate` event that requires a response.

TRANSMIT.confirm

This service primitive is used by the network layer to return the status of a previously triggered `TRANSMIT.request` back to the caller.

FLUSH.request

This service primitive is used to delete a packet by handle.

FLUSH.confirm

This service primitive is used to indicate whether a packet was successfully deleted or not.

LOCAL_MANAGEMENT.request

This service primitive is used to configure properties on the Network Layer.

LOCAL_MANAGEMENT.confirm

This service primitive is used to return the results of a `LOCAL_MANAGEMENT .confirm` to whomever executed the request.

LOCAL_MANAGEMENT.indication

This service primitive is used to notify about a Network Layer event which was not requested.

3.4.3 Routing

Routing in WirelessHART defines the type of routing a device uses to forward a packet The WirelessHART network layer is responsible for forwarding packets and the standard allows two types of routing which need to be supported by all devices, namely graph routing and source routing. When it comes to routing protocols used to make these routes, the WirelessHART standard does not define which protocol the Network Manager should use, this is left up to the developers.

3.4.3.1 Graph Routing

In graph routing a device is called a node and if two nodes can communicate with each other they are connected through a link. When graph routing is used in the forwarding of a packet, a Graph ID is given and optional source routing has not been indicated in the NPDU, as described earlier in this chapter. A device has several graphs and each graph gives the device an option of one or more neighbors to forward a packet to, in order to get the packets to their final destination.

In WirelessHART these graphs are created and maintained by the Network Manager and each field device has its own list of graphs it can use to route to any member of the network. Each of these graphs are identified by a Graph ID, which is included in the NPDU of a packet. This ID tells a forwarding device which graph it should use for routing and there will be a choice of one or usually several neighboring nodes to choose from.

Since graph routing usually gives each node several neighbors to choose from and the Network Manager dynamically maintains the graphs it is reliable even if nodes fail or disappear. Figure 3.12 shows an example of graph routing. If a node has several links the packet can be forwarded on any of them. Due to this packets going from A to H will not necessarily take the same path each time.

WirelessHART also supports a simplified version of graph routing called superframe routing. When this is used the Graph ID field is set to a superframe ID and when a device tries to look up the ID in its list of

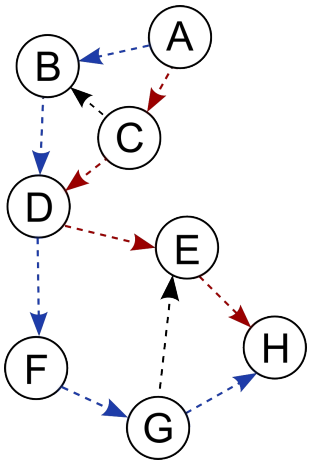


Figure 3.12: Graph Routing - Example of a graph uniquely identified by a Graph ID. A packet is to be routed from A to H. A node with more than one link can choose which node it wishes to forward the packet to. In case of the node fails the forwarding node may forward the packet through another neighboring node there are neighbors available. The image shows examples of two routes: the red one and the blue one. Both these paths are allowed and each node chooses which neighbor it wants to forward the packet to. The remaining black arrows are links which are not in the two paths, but are possible to use.

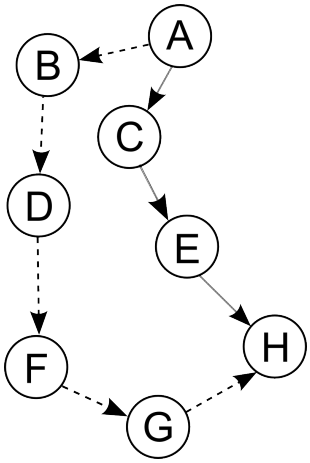


Figure 3.13: Source Routing - A packet is to be sent from A to H. The figure shows two different instances of source routing. In one case the source route indicated is <B, D, F, G> and in the other case it is <C, E>.

graphs it will fail and then assume it is a superframe ID in stead. If the superframe ID is valid the Network Layer will look for a neighbor with a link in that superframe and forward the packet to it.

3.4.3.2 Source Routing

The alternative to graph routing in WirelessHART is source routing. When the control byte source routing bit is set the device is to use the given source route list found at the end of the NPDU header. The source route list is set by the Network Manager and each time a node forwards a packet it looks the route up in the list and sends it to the next node. Since the list only gives each node one option of which other node to forward the packet to it is prone to node failure. If any node in the list fails there is no alternative route and the packet will be dropped. Source routing is described by the WirelessHART standard as something that is only to be used for testing and debugging paths and not for normal packet routing. Figure 3.13 displays two examples of source routing.

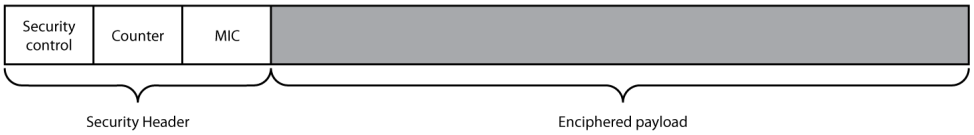


Figure 3.14: Network Layer Protocol Data Unit Security Header

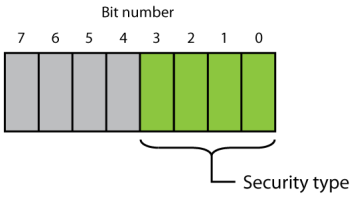


Figure 3.15: The Security Control Byte

3.4.4 Security Sub-layer

In addition, on the network layer there is a security header which is used for encapsulating enciphered data, this header is displayed in figure 3.14. An overview of the fields and their lengths are shown in table 3.4.

Position	**Name/Description**	**Length in Bytes**
1	Security Control Field	1
2	Nonce Counter	1 or 4
3	Message Integrity Code (MIC)	4

Table 3.4: Security Sub-Layer PDU

Security Control Byte

The security control byte is used to determine what type of security the NPDU payload is enciphered with. Figure 3.15 shows which options are available and the type of security decides the size of the nonce counter field.

Nonce Counter

Depending on the control byte the nonce counter can have 2 different

lengths; 1 byte if the security type is session keyed and 4 bytes if it is join or handheld keyed.

Message Integrity Code (MIC)

The MIC field contains a value which is used to check the integrity of the transmitted frame. If the value calculated at the receiver side does not match the value set in this field the frame has been tampered with or there has been an error in the transmission. The MIC value is calculated using a MIC algorithm using a shared key which both the sender and receiver knows.

Enciphered Payload

The payload of the security sub-layer is enciphered in order to prevent intermediate devices from reading it. The payload contains the Transport Layer PDU.

3.5 Network Components

In this section we provide an overview of the various network components that are associated with a WirelessHART environment, starting out with the components that are most relevant to this project, namely the Field Devices, Gateway, Virtual Gateway, Access Point and Network Manager. After which we continue to describe a few other components which are not directly relevant to this project but still commonly seen in a WirelessHART installation.

3.5.1 Field Devices

Field devices are devices that integrate wireless functionality into HART devices. They connect to the WirelessHART network, each device is able to act as both a source and a sink for packets, in addition to be able to route packets in the network. These devices are intended to be connected to process control equipment, and make up the most of the packet data

in a network. They are also required to be able to do intermediate packet routing.

3.5.2 Gateway

The overall task of the Gateway is to enable communication between the host applications in the WirelessHART network, every WirelessHART network contains a Gateway. The Gateway in a WirelessHART network is a logical unit that contains the Virtual Gateway, Network Manager, Security Manager and any given number of Access Points. In addition to this it also describes a host interface to the production backbone, this being either for example Ethernet.

3.5.2.1 Virtual Gateway

The primary responsibility of the Virtual Gateway in the WirelessHART network is to enable communication between the network manger and the rest of the network. The Gateway functionality is divided into a Virtual Gateway and one or more Access Points. The standard defines that each WirelessHART network must have one and only one Gateway. This Gateway can consist of only one Access Point or several, depending on the demand. Several Access Points increase the throughput and reliability of the network as several Access Points can send and receive data to and from the Gateway. The standard also defines that the Gateway needs to have a well known address which all network devices have to know. This address consists of the Unique ID: `0xF981 0x000002` and the device nickname `0xF981`.

3.5.2.2 Access Point

This is the device that handles the translation from WirelessHART communication to any given transport media that the network administration entities use between each other. This can typically be a gateway between WirelessHART and IEEE 802.3 (Ethernet). But it can also be a part of one application where the Access Point is separated logically as a class.

3.5.2.3 Network Manager

The Network Manager (NM) responsibility is managing the network, this includes scheduling communication, keeping track of security information and the overall state of the network. There is always only one active Network Manager in the WirelessHART network at any given time. It maintains a complete list of all network devices and governs parts, joins and routing schemes between these. A subordinate feature is the collection and maintaining of various performance metrics related to the network. It also monitors these metrics and can report and/or instigate countermeasures if certain thresholds meet or fail to meet requirements. The Network Manager itself communicates over a secure channel with the Security Manager and the network Gateway. It is addressed with the short name `0xF980`.

3.5.2.4 Security Manager

In any given WirelessHART network, join, session and network keys shall be provided to ensure adequate encryption to both data transfer and device authentication. These keys are to come from a centralized entity that provides this to either the entire network, or more than one WirelessHART network via the Network Manager. A Security Manager can also support external applications where this is applicable.

3.5.3 Production Backbone

The production backbone or plant automation network is the network which the Gateway is connected to. Normally this network would constitute a regular Ethernet network, but there is also the possibility that this network is also wireless. The Gateway is responsible for bridging communications between the WirelessHART network and the production backbone.

3.5.4 Handhelds

Handhelds in the context of WirelessHART are devices which are portable WirelessHART-enabled computers containing host applications that are used to configure, diagnose and monitor the WirelessHART network. One of the primary differences between a node and a handheld device is that the latter is not required to support routing.

3.5.5 Wireless Adapters

A wireless adapter is used to join existing HART equipment to the WirelessHART network. The adapter is serving as a bridge between the wired HART equipment and the wireless network.

3.5.6 Host Applications

Host applications are applications that are running on computers that are used to manage, diagnose and monitor the network. These applications include functionality to communicate with the WirelessHART network through the Gateway.

3.6 Time Synchronization

Since WirelessHART utilizes TDMA in its operation, all the nodes in the network need to have the same concept of the current time.

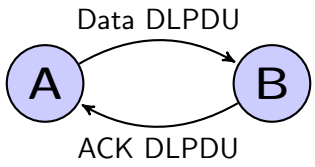
In the current revision of the standard, the IEEE 802.15.4-2006 it defines that for example the `TsTxOffset`, which is the offset between the beginning of a slot and to the point in time the transmission can start, can not exceed +/100 μsec.

According to [31] there are two major factors that affect the accuracy of the local system clock, namely clock drift and clock offset. We define clock drift as the change in frequency of the clock over time. A change of frequency can have several reasons including imperfections in the crystal and temperature changes. Clock offset can be defined as the relative offset

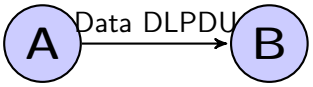
from the "real" time. In the case of our WirelessHART network we define the "real" time as the local system time of the Network Manager.

There are two primary one-hop methods of synchronization on the MAC layer as reported in the literature [27]. These methods are named Active and Passive synchronization. In addition we add one definition which we name Initial synchronization which denotes the state where the node has no prior knowledge of the network.

Each of these states will be explained in the following sections.



(a) Active synchronization



(b) Passive synchronization

Figure 3.16: Time synchronization

3.6.1 Initial Synchronization

Initial synchronization is where the node tries to join the WirelessHART network. When a node tries to join the network it has no information about the network other than the PAN ID. This means that the first key is to get properly synchronized with the rest of the network on the MAC layer. The way to accomplish this is to listen for data packets and adjust the internal clock to match the slot timing. After the node has gotten the proper slot timing, radio usage can be limited to only slot times. This is a form of passive synchronization because the node just sits there and listens for packets that it can use for calibrating the internal system clock.

3.6.2 Active Synchronization

In active synchronization[31, 2] (Figure 3.16) the source node (A) will start to send the packet at exactly `TsTxOffset` according to its own system clock. When the destination node (B) has completely finished receiving the packet it will note the time defined as t1. After t1 has been calculated node B may calculate the relative time difference between the time the packet was received according to its local system time and when the packet should have been sent by the source node in a perfect world. The adjustment needed using the destination node as a reference can in turn be computed and is sent back with the ACK as a signed integer. The source node may then use this information to adjust its own clock if B is marked as a time source node in the superframe.

$$T_{adj} = TsTxOffset - t1$$

3.6.3 Passive Synchronization

In passive synchronization mode[31, 2] (Figure 3.16), the node that is being adjusted is the destination and not the source. The source node sends the message exactly at `TsTxOffset` just like in active synchronization and the destination notes the time it finished receiving the packet as t1. The difference is that the relative time difference is not sent back to the source node in the ACK but instead it is used by the destination node to adjust its own internal system clock.

$$T_{adj} = t1 - TsTxOffset$$

3.7 WirelessHART Join Process

When a new device is to join an already existing WirelessHART network a join process needs to be followed in order for the device to be properly integrated in the network. Most of the join process is to ensure the integrity of the newly joining device in order to prevent malicious devices to join and

possibly harm the network. In the following section we will explain how the WirelessHART join process works by giving an overview of what has to happen and then a more detailed view of what goes on in the Network Layer (NL) and the Data Link Layer (DLL) during the join process. An overview of the WirelessHART join process can be found in figure 3.17.

3.7.1 Overview

The standard describes the join process as a sequence of 7 events:

3.7.1.1 Preparing the New Device

The new device that wishes to join the network first needs to be provisioned with the Network ID it is to join as well as the join key it has to use in order to authenticate with the Network Manager.

3.7.1.2 Listening for Packets

After obtaining the Network ID and join key the new device enters search mode and starts listening for other devices. It tries to capture advertise packets from future neighbors, but also tries to capture normal DLPDU packets in order to synchronize to the network. While doing this the new device will create statistics of the packets it receives which will later be used when communicating with the Network Manager. When the desired amount of advertise packets has been received and the new device is synchronized to the network it selects a neighbor with an adequate Receive Signal Level to send its join request through.

3.7.1.3 Presenting Credentials

After having received enough advertise packets and being synchronized, the new device creates a join request, enciphered with the join key, and sends this to the Network Manager. The packet contains all the information the Network Manager needs in order to authenticate the new device and examines these upon the arrival of a join request. In order to continue

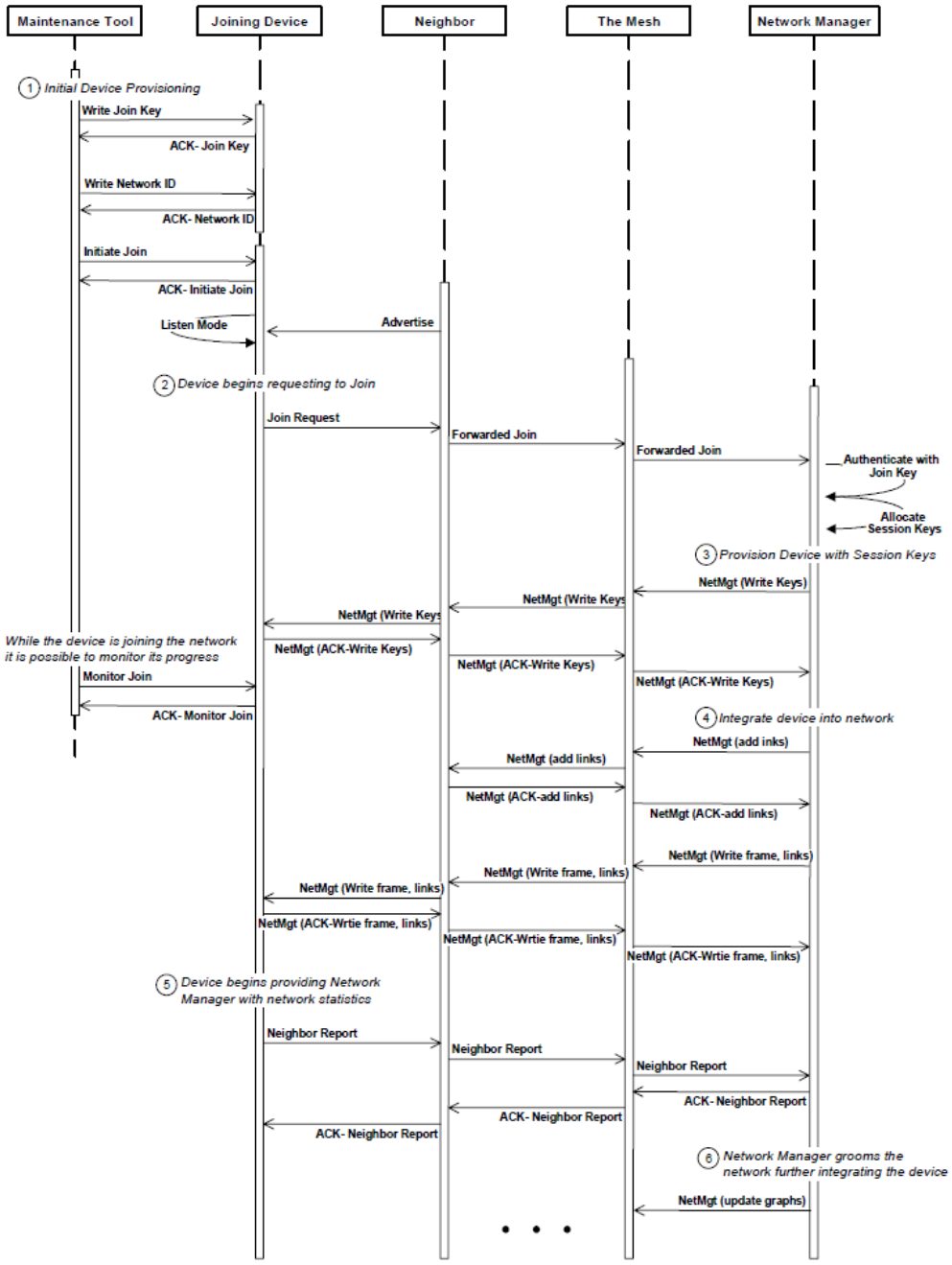


Figure 3.17: An overview of the WirelessHART Join Process[29, Figure 41]

communication with the Network Manager the new device needs a session key and a network key, which the Network Manager provides given that the device has given:

1. Its identity
2. Its long tag
3. The list of neighbors detected while waiting for advertise packets
4. The correct join key

If these are given and the join key is correct the Network Manager can now send the new device a session key and a network key. If this, however, did not happen before the join request timeout another join request is sent to the Network Manager until maxJoinRetries has been reached.

3.7.1.4 Receiving the Keys

If the joining device has a trusted identity, is using the right join key and properly combines its name and password the new device and the join request will be authenticated. To signal this, the Network Manager allocates session keys, the new device's nickname and the network key and sends all of these back to the new device, still enciphered with the join key. The new device now sends an acknowledgment packet to the Network Manager using the new session and network keys.

3.7.1.5 Joining the Network

Although the new device has been authenticated it has still not fully joined the network. At the moment the device is only allowed to communicate with the Network Manager through the route it chose the when sending the Join Request. In order to fully join the network the Network Manager needs to further integrate the device into the network by providing it with at least two time-source parents, updating the communication tables in the device's parents (neighbors) and transferring the new device's communication from join links to normal links. This achieved by the Network

Manager sending a series of frames to the new device with this information and more and the new device has then successfully joined the network.

3.7.1.6 Quarantine

Although the device has joined the network the Network Manager can choose to leave it in quarantine for a long as it wishes in order to ensure that the device is trustworthy. While in quarantine the device does not have any sessions with the Gateway, can only communicate with the Network Manager and is therefore not allowed to publish any process data. In addition to this the device generate Health Reports that it periodically sends to the Network Manager, which among other contains neighbor statistics much like the ones created in the second event. When the Network Manager is convinced the new device is trustworthy it gives the device a Gateway session and so ends the quarantine. Depending on the Network Manager's security strategy it can also just immediately give the new device a Gateway session, skipping the quarantine altogether.

3.7.1.7 Obtaining a Gateway Session

As soon as the new device has received a gateway session it becomes operational and is now a full-worthy member of the network. The Gateway will now begin using the maintenance service to communicate with the new device filling its data-cache and consuming a high amount of bandwidth while doing so. Then the device operates as all the other nodes and the join process has ended.

3.7.2 The Network Layer Join Process

During the join process the Network Layer (NL) and the Data-Link Layer (DLL) on the device work closely together to ensure a successful joining attempt. Both the NL and the DLL have their own join procedures which are managed by two state machines, one for the NL and one for the DLL. We will begin with the Network Layer join process and its state machine

(Figure 3.18) and then explain the Data Link Layer join process and state machine in the next section.

3.7.2.1 Network Layer States

Searching

While in search mode the NL basically only waits for the DLL to find the network, synchronize to it and receive an advertise packet. When this happens the DLL sends an `ADVERTISE.indicate` signal which lets the NL switch to the Got an Advertisement state.

Got an Advertisement

When entering the Got an Advertisement state the NL starts the `AdWaitTimer` and waits for more advertise packets. It continues to do so either until the desired number of different advertise packets has been received or until the AdWaitTimer reaches `AdWaitTimeout`, which is how long the NL will wait for advertise packets. It then moves on to the Requesting Admission state.

Requesting Admission

When entering this state we have come to the part where the device sends a Join Request and then waits for a response. When sending a Join Request the `JoinRspTimer` is started and if it reaches `JoinRspTimeout` the Join Request failed and another is sent until the maximum of retries has been reached. If this happens the join has failed and the state machine exits with an error. If the Join Request succeeds the Loosely Coupled state is initiated.

Loosely Coupled

Since the Join Request succeeded the device now has a nickname, a Session Key and a Network Key and is now loosely coupled to the network. This means that it can only communicate with the Network Manager and has still not received a normal frame and links. When this happens the join has been a success and the state machine terminates with no error.

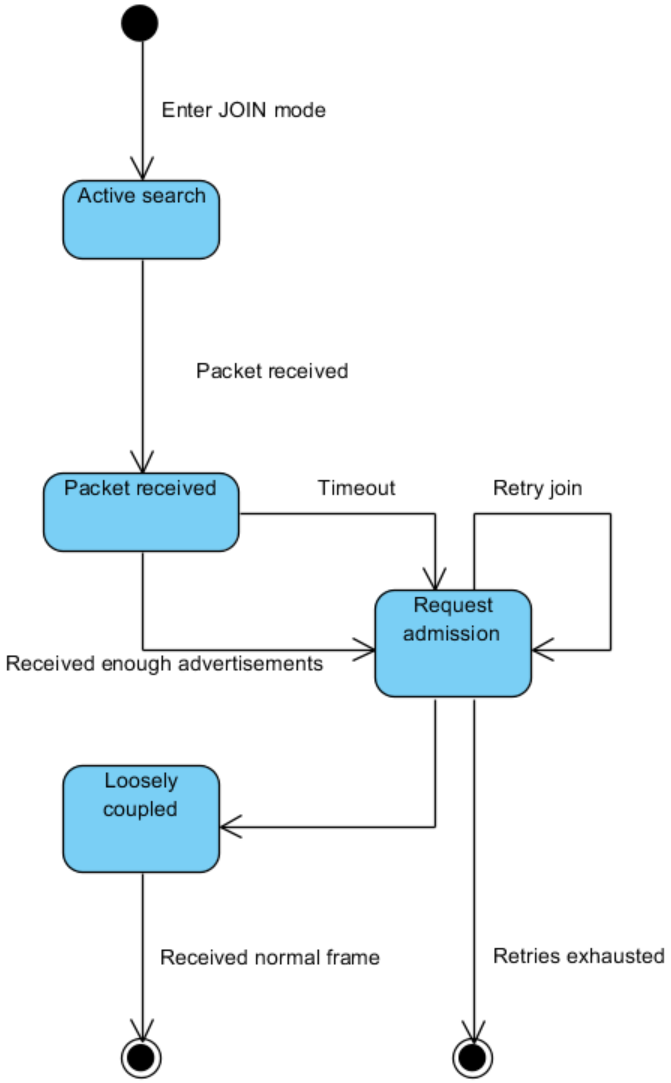


Figure 3.18: The Network Layer Join Process State Machine

3.7.3 The Data Link Layer Join Process

This section describes what happens in the Data Link Layer (DLL) during the WirelessHART join process by looking at the DLL state machine (Figure 3.19).

3.7.3.1 Data Link Layer States

Active Search

When the join process starts the DLL enters the active search state which tells the new device to actively listen for packets. It will actively listen for `ActiveSearchShedTime` and listens to each channel for ChannelSearchTime in order to scan all the channels for incoming packets. If `ActiveSearchShedTime` is reached without finding the network (i.e. getting a packet) Active Search ends and the Passive Search state is reached. If it does receive a packet, thus discovering the network, the Packet Received state will be initiated.

Passive Search

The Passive Search state can be reached by either Active Search or Packet Received as a result of the `ActiveSearchShedTime` timing out. Passive search mode is a power-saving search where instead of listening all the time the device listens for PassiveWakeTime and then enters low-power mode, turning of the transceiver, and then listens for a while again. This is repeated every `PassiveCycleTime` until a packet is received which brings it to the Packet Received state.

Packet Received

When a packet with the right Network ID is received this state is entered. The goal of this state is to synchronize the device with the network by recording the start time of all non-ACK DLPDUs received in the network. The device gathers statistics about its neighbors and slot times and is considered synchronized when the slot timing statistics coincide. While in this state the device will also update its neighbor table and if it receives an advertise packet it

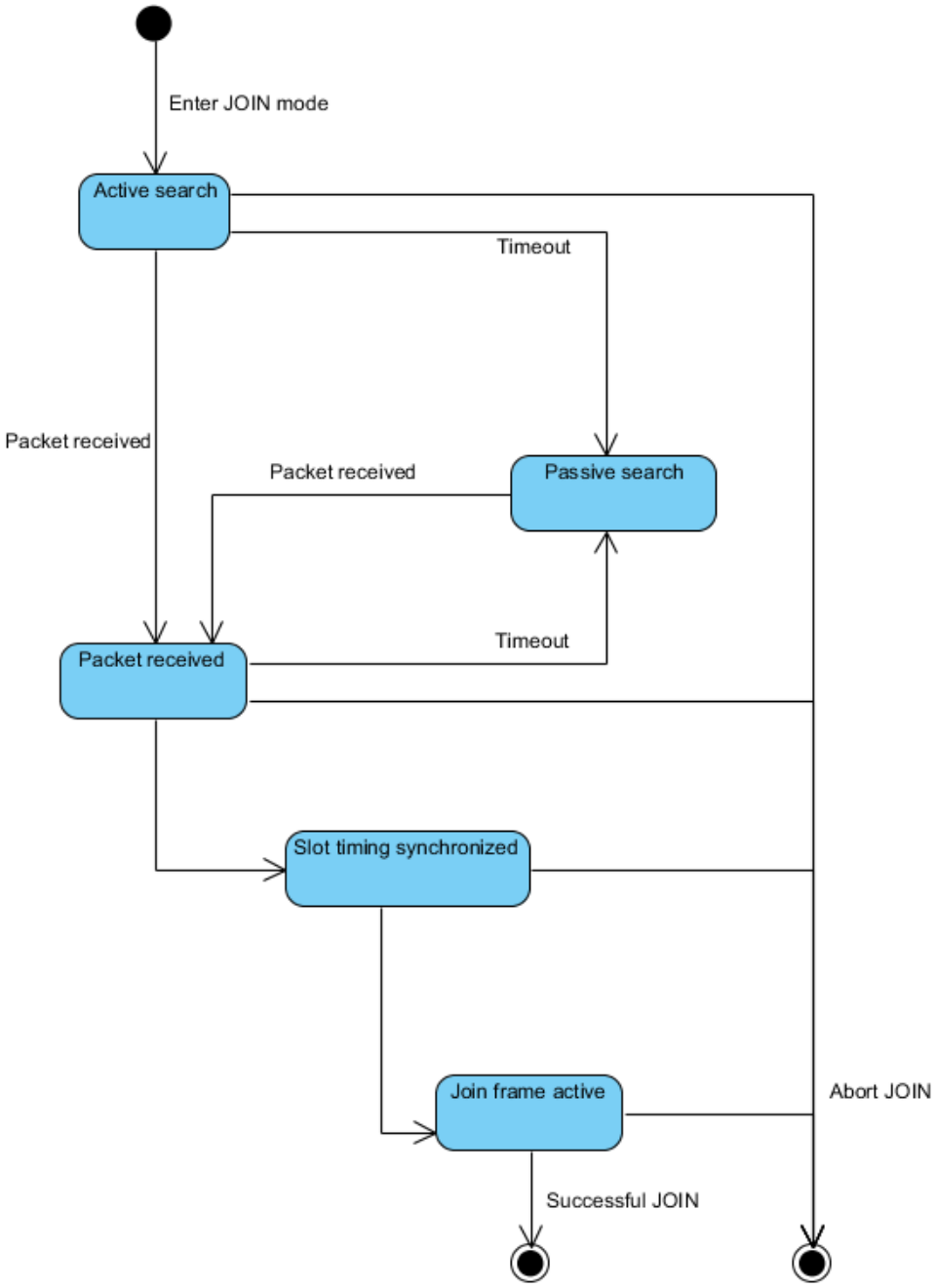


Figure 3.19: The Data Link Layer Join Process State Machine

will continue only searching the channels that are indicated by the advertise packet's channel map. When the slot time is synchronized and the absolute slot time is aligned to the network the Slot Time Synchronized state is entered.

Slot Time Synchronized

Since the slot times are now synchronized the DLL only needs to listen for packets in slots, as oppose to constantly listening for packets. To leave this state the DLL waits for the NL to send a Join Request, however this will not happen until the NL has received an `ADVERTISE.indicate` from the DLL. Once the DLL receives an advertise packet this `ADVERTISE.indicate` is sent to the NL and the DLL will initialize the graph received in the advertise packet and connect itself to the advertising device. As soon as the DLL receives a Join Request from the NL it will enter the Join Frame Active state.

Join Frame Active

Since join slots are shared there is a chance other new devices will try and use the same slot to send their Join Request. In order to try and avoid collision the Back-Off Exponent (`BOExp`) shall be set to 4 before sending the Join Request. When the Join Request has been sent the DLL will issue Keep-Alive packets, using Join Links, in order to stay synchronized with the neighbors. This state will be kept until the join either succeeds or fails.

3.8 Chapter Summary

Throughout this chapter we have examined how the WirelessHART standard compares to the layering in the standard OSI model and how the layers are designed bottom-to-top.

We have provided an in-depth look at the WirelessHART specification including the individual protocol layers and the physical requirements defined by it. After describing the physical layer and the physical characteristics of the communication medium we moved on to the link layer

and explained how it is sub-layered into the LLC and MAC layer. Finally we gave an introduction to the design and specification of the network layer. As previously stated, the transport layer and higher layers in the OSI model have not been discussed as they are not directly relevant to this project.

At the end of the chapter we have provided a short introduction to the various types of network components defined by the WirelessHART standard, discussed the basic issues and theory for time synchronization in addition to an overview of the WirelessHART join process both on the link layer and the network layer.

Chapter 4

Hardware and Software

In this chapter we have provided an overview of the hardware and software utilized in our development and testing environment including which decisions have been made when selecting software and hardware to work with.

4.1 Hardware

In this section we have provided a brief introduction to each of the hardware devices available to us during this project, including the hardware that we had available but did not use. In addition we have provided some reasoning behind our choices of hardware.

4.1.1 AVRRZRaven (ATmega1284P)

The AVRRZRaven kit (Figure 4.1) is an evaluation kit for the development of wireless network applications on the ATmega1284P Micro controller Unit (MCU). It consists of two AVRRZRaven (Raven) boards and an AVRRZRavenUSB stick (RavenUSB). Evaluation kits are circuit boards that are created for development and evaluation purposes, it contains the micro controller and the necessary support logic to be able to interface and develop on the boards.

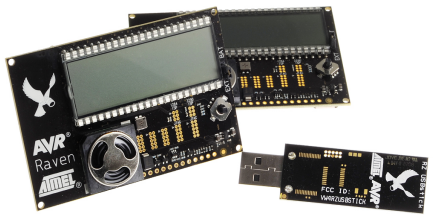


Figure 4.1: Contents of the AVRRZRaven kit

Each Raven board is equipped with a 2.4GHz radio, two micro controllers and a LCD screen. The logical structure of the ATmega1284P is displayed in figure 4.2. The micro controller to the left (ATmega3290P) is responsible for the operation of peripheral hardware such as the LCD screen, speaker, sensors, joystick, interfaces, etc. The micro controller to the right (ATmega1284P) is the controller for the radio and this is the controller we focus on during the implementation of this project. The radio itself is an AT86RF230B chip. Figures 4.3 and 4.4 give an overview of the assembly of the AVRRZRaven board.

The RavenUSB stick is basically just a 2.4GHz transceiver stuck on a board with an USB connector. The micro controller used on this board is the AT90USB1287. Refer to figure 4.5 and figure 4.6 for an overview of the assembly of the RavenUSB stick.

4.1.1.1 Supported Clock Sources

By looking at the AVR RZ Raven boards in detail we have discovered that the board supports the following clock sources

- Low Power Crystal Oscillator

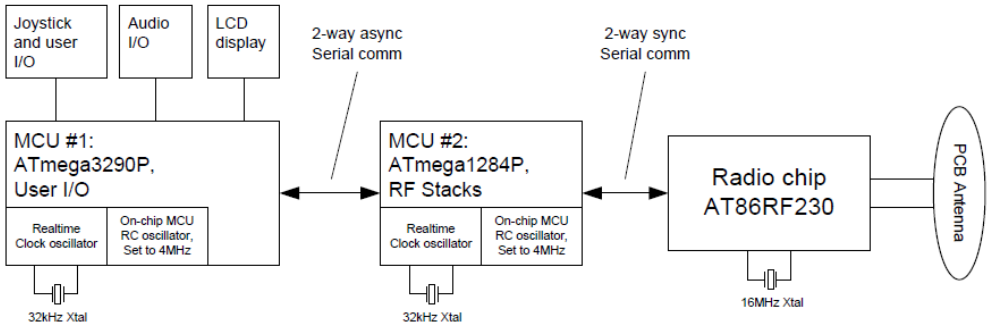


Figure 4.2: AVRRZRaven Block Diagram showing the logical separation of the different micro controllers and major components on the board.

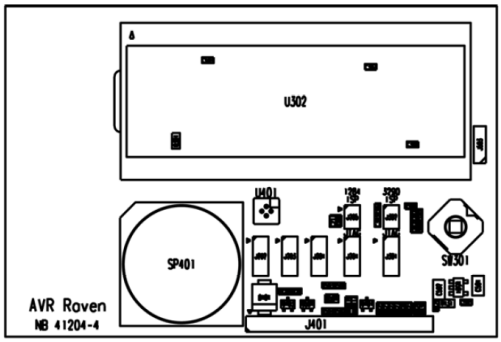


Figure 4.3: AVRRZRaven Front Assembly

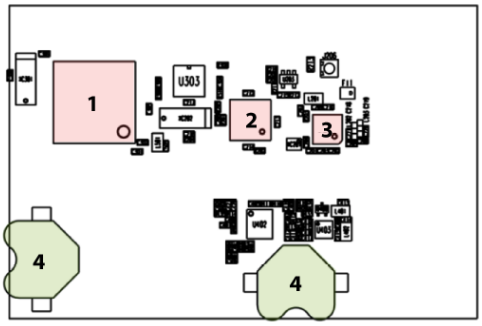


Figure 4.4: AVRRZRaven Back Assembly - Showing the primary components: 1. ATmega3290, 2. ATmega1284P, 3. AT86RF230B

- Full Swing Crystal Oscillator
- Low Frequency Crystal Oscillator

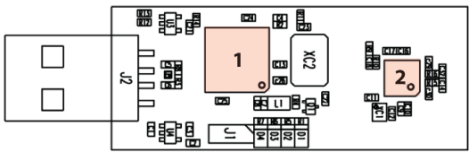


Figure 4.5: RavenUSB Front Assembly - Showing the primary components: 1. AT90USB1287, 2. AT86RF230B

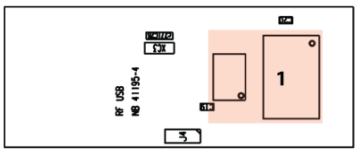


Figure 4.6: RavenUSB Back Assembly - Showing the primary components: 1. Memory chips

- Internal 128 kHz RC Oscillator
- Calibrated Internal RC Oscillator
- External Clock

The default clock source among these is the internal 128kHz RC oscillator. This is mainly due to their ability to draw little power during sleep periods. This RC oscillator is inherently inaccurate by design in addition to a susceptibility to temperature changes.

4.1.2 ATmega128RFA1

The ATmega128RFA1 (Figure 4.7) is a relatively new micro controller we received from Atmel which is a single-chip solution based on ATmega1281. Single-chip solution means that all the hardware such as the micro controller and radio is self-contained in a single chip. One of the most prominent features of this micro controller is the hardware support for AES encryption, in addition the chip has been designed for very low power consumption. The sensitivity, data rate and output power of the radio in this chip is approximately the same as the AT86RF230B which is the

radio that ships on the Raven kit. During the course of this thesis, we have also investigated the effort required to migrate the implementation to this new MCU. Also here, the work required depended heavily on the amount of change that had been made to the original AVR2025 library and how much of these changes were transferable without modifications to the library itself.

Figure 4.7: Two ATMEGA128RFA1s and antennas

4.1.3 STK541

The STK541 (Figure 4.8) is an USB adapter board manufactured by Atmel which can be used as a wireless sniffer in co-existence with the Daintree Sensor Network Analyzer described in section 4.2.2, in order to verify the implementation. During initial testing of the project we had a lot of issues with this device freezing and becoming very unstable. Due to this we chose to move away from this device and instead try to utilize the Raven USB Stick as a sniffer in conjunction with Wireshark, this process is more thoroughly described in section 4.2.3.1 and 4.2.5.

Figure 4.8: STK541

4.1.4 STK600

The STK600 (Figure 4.9) is a standard form factor socket card for doing development on single chip solutions connected on small router boards. Single chip solutions simply means that all the logic and controllers are embedded in a single chip as opposed to for example the RZ Raven boards where the radio and micro controller are separate chips connected on a circuit board. This kit will allow us to power and program the new ATmega128RFA1 MCU. In addition the STK600 comes with another MCU which is an ATmega2560. We did not devote very much time looking at this new hardware at this stage in the project, because we felt it was more important to get an overview of the existing implementation and how it worked on the hardware it was written for.

4.1.5 JTAGICE mkII

The JTAGICE mkII is a debugging tool (Figure 4.10) for AVR micro controllers as well as for any other chip that supports JTAG. It supports on-chip debugging and compared to its predecessor the JTAGICE mkI it supports connections through USB. It supports In-System Programming

Figure 4.9: STK600

(ISP) through 6 or 10 pin adapters and for communication it uses a variant of the STK500 protocol[24].

ISP allows us to program the micro controllers while they are installed in a complete system instead of having to program them before installing them into the system. This means the programming and testing can be done in one step instead of having to remove the micro controller from the chip, program it, reassemble the chip again and then test it, which of course saves a lot of time.

4.1.5.1 On-chip Debugging

The JTAGICE mkII (ICE - In Circuit Emulator) enables us to do on-chip debugging of our code on the AVR micro controllers. It is used together with the AVR Studio's user interface and makes debugging of the code a lot simpler. The JTAGICE emulates the micro controller without having to remove it from the target system (the node) and allows us to use break points and step the code. It also allows the monitoring of all AVR resources, examples of these being flash memory, SRAM, register files and I/O modules. One of the primary down-sides with using on-chip debugging reveals itself when debugging multiple nodes. If a node is connected to the debugger it runs at a much lower clock frequency, while the MCU by default runs at 8MHz the maximum clock frequency the debugger can keep up with is 1/4th of that. This makes debugging timing issues and issues that require multiple nodes to communicate very difficult and has proven one of the primary challenges during this project.

Figure 4.10: JTAGICE mkII

4.1.6 Choosing between ATmega1284P and ATmega128RFA1

As we, in addition to the ATmega1284P micro controllers, received a couple of newer ATmega128RFA1 micro controllers from Atmel, we needed to evaluate the work involved in migrating the code to the newer micro controller. The first step in this process was to try updating the AVR2025 library to the newest version, 2.7.0 at the time of writing, (more on the AVR2025 library in section 4.2.6) in order to get support for the ATmega128RFA1's radio in the Transceiver Abstraction Layer. Since the implementation of this project has undergone some major modifications in terms of the original AVR2025 software library, especially in the MAC layer, we needed to carefully consider the implications of moving towards a newer version. This includes how much of the code will need to be rewritten in order to build a more detached interface between the implementation of the WirelessHART stack and the AVR2025 library in order to have smoother upgrades in the future.

4.2 Software

This section provides an introduction to the software packages utilized for development, packet-sniffing and debugging of the project including software that we originally intended to utilize but have abandoned for reasons explained in the text.

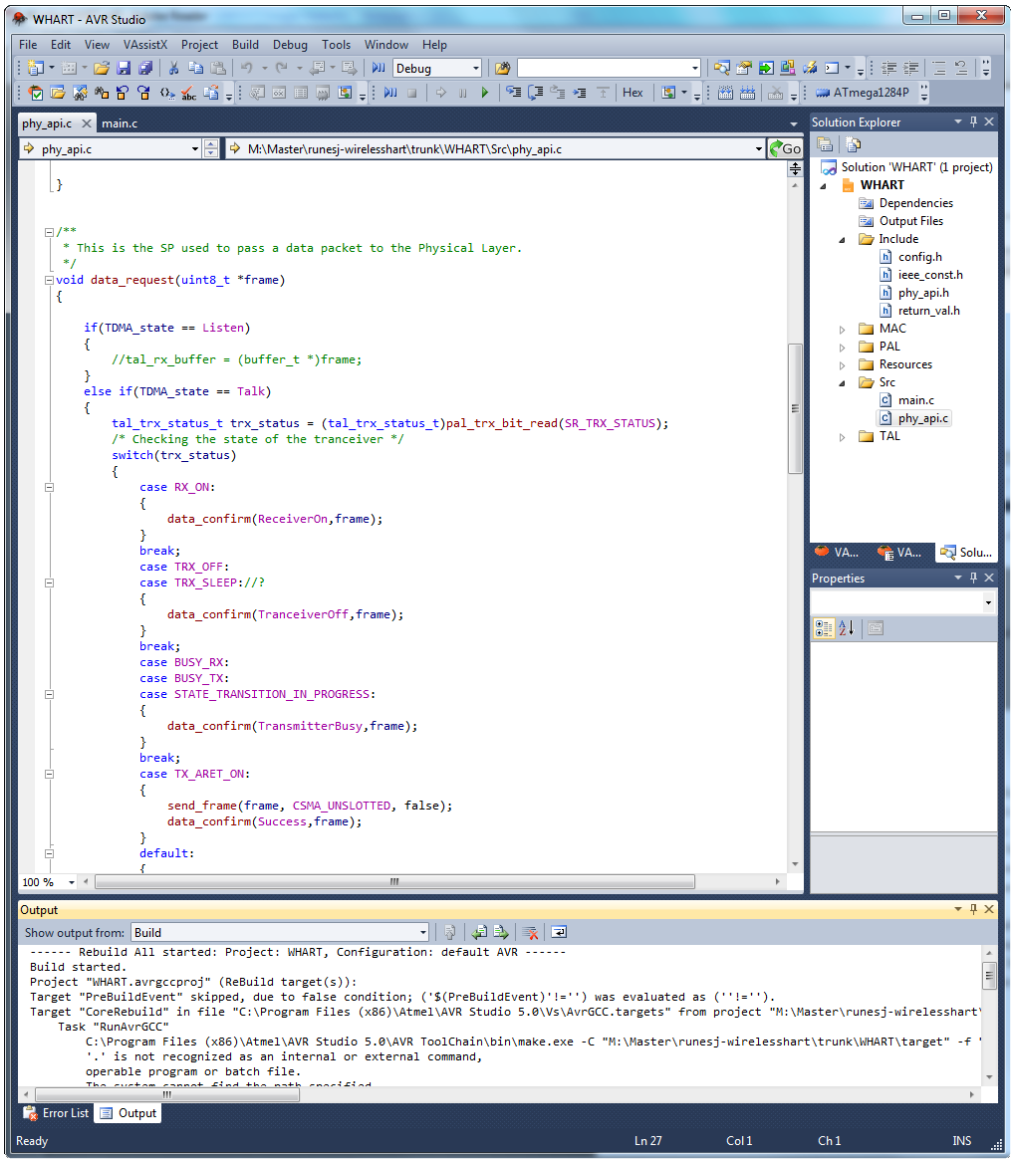


Figure 4.11: The AVR Studio 5 IDE

4.2.1 AVR Studio

AVR Studio is an Integrated Development Environment (IDE) for developing and debugging embedded application on AVR chips. The IDE supports on-chip debugging, breakpoints, viewing of registers, I/O ports and instruction level stepping. This makes it a very powerful tool for the development of embedded AVR applications. AVR Studio 5 is the newest version and is built on Visual Studio which gives it a powerful base in terms of editor features compared to the previous version. At the time of writing it is recently out of beta, but it seems generally stable and better than the previous version, since it has the same advantages plus new features. Some of the most significant features added compared to AVR Studio 4 is the code helpers which add code completion and outlines to the environment making it a lot easier to navigate in larger projects. A screen shot of the AVR Studio 5 IDE can be seen in figure 4.11.

4.2.2 Daintree Sensor Network Analyzer

Powerful debugging is a great advantage, especially when developing software for embedded chips and micro controllers where the debugging possibilities are usually limited. The Daintree sensor network analyzer allows us to capture the traffic on a certain channel within the 2.4GHz radio band and to decode this captured traffic as 802.15.4 traffic. Although this is not a complete decode in terms of WirelessHART it is close enough as it gives us a lot of information about the 802.15.4 frame headers including source and destination addresses. Figures 4.12, 4.13 and 4.14 show examples of the Daintree Sensor Network Analyzer software sniffing WirelessHART packets.

4.2.2.1 Issues

We ran into several problems when attempting to set up and use Daintree. The first issue was getting a license for the software when installing it and while trying to obtain one we found out the software had been discontin-

ued. After contacting Atmel they provided us new discs with the Daintree software and licenses so we were able to install it eventually. In addition to this the software was only available for installation on Windows XP and it would frequently crash for no apparent reason. This is when we decided to look at alternatives for sniffers, which will be further described in section 4.2.3.

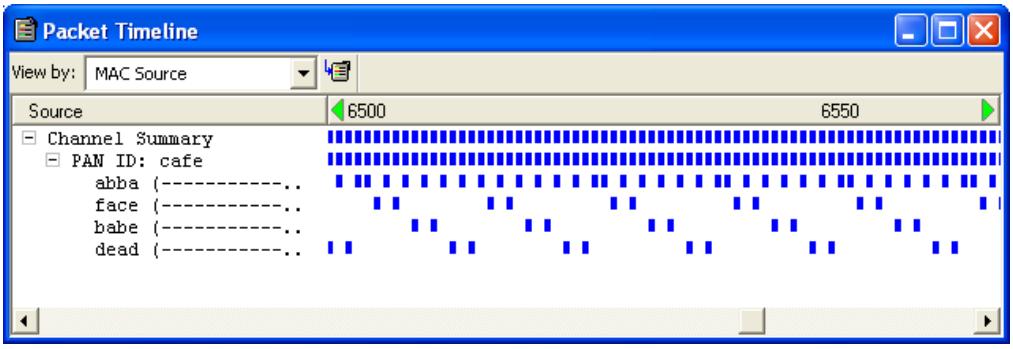
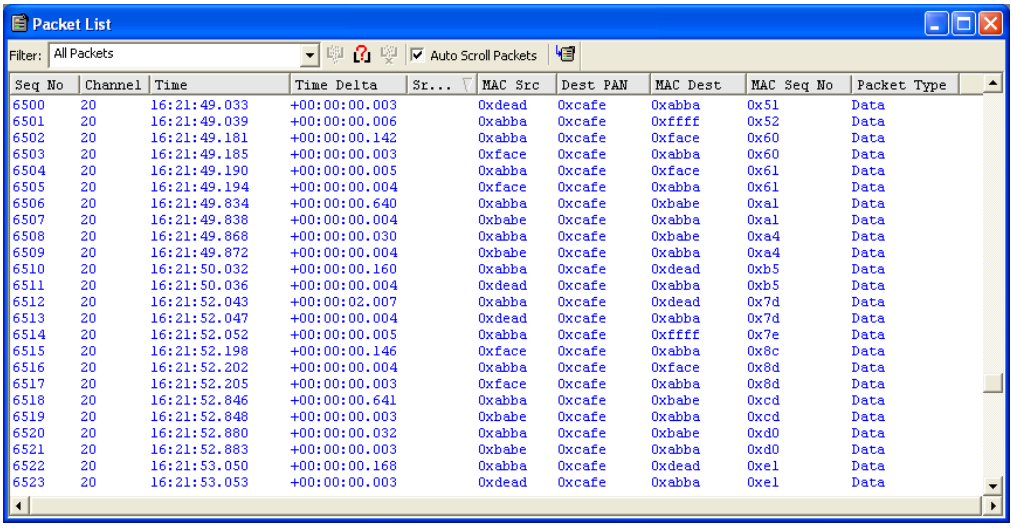


Figure 4.12: The Daintree Sensor Network Analyzer Block View provides an excellent overview over the communication happening between multiple nodes. In this figure there are 4 nodes communication on the same network and the blocks represent packets that are transmitted between these nodes.



Seq No	Channel	Time	Time Delta	Sr...	MAC Src	Dest PAN	MAC Dest	MAC Seq No	Packet Type
6500	20	16:21:49.033	+00:00:00.003		0xdead	0xcafe	0xabba	0x51	Data
6501	20	16:21:49.039	+00:00:00.006		0xabba	0xcafe	0xffff	0x52	Data
6502	20	16:21:49.181	+00:00:00.142		0xabba	0xcafe	0xface	0x60	Data
6503	20	16:21:49.185	+00:00:00.003		0xface	0xcafe	0xabba	0x60	Data
6504	20	16:21:49.190	+00:00:00.005		0xabba	0xcafe	0xface	0x61	Data
6505	20	16:21:49.194	+00:00:00.004		0xface	0xcafe	0xabba	0x61	Data
6506	20	16:21:49.834	+00:00:00.640		0xabba	0xcafe	0xbabe	0xa1	Data
6507	20	16:21:49.838	+00:00:00.004		0xbabe	0xcafe	0xabba	0xa1	Data
6508	20	16:21:49.868	+00:00:00.030		0xabba	0xcafe	0xbabe	0xa4	Data
6509	20	16:21:49.872	+00:00:00.004		0xbabe	0xcafe	0xabba	0xa4	Data
6510	20	16:21:50.032	+00:00:00.160		0xabba	0xcafe	0xdead	0xb5	Data
6511	20	16:21:50.036	+00:00:00.004		0xdead	0xcafe	0xabba	0xb5	Data
6512	20	16:21:52.043	+00:00:02.007		0xabba	0xcafe	0xdead	0x7d	Data
6513	20	16:21:52.047	+00:00:00.004		0xdead	0xcafe	0xabba	0x7d	Data
6514	20	16:21:52.052	+00:00:00.005		0xabba	0xcafe	0xffff	0x7e	Data
6515	20	16:21:52.198	+00:00:00.146		0xface	0xcafe	0xabba	0x8c	Data
6516	20	16:21:52.202	+00:00:00.004		0xabba	0xcafe	0xface	0x8d	Data
6517	20	16:21:52.205	+00:00:00.003		0xface	0xcafe	0xabba	0x8d	Data
6518	20	16:21:52.846	+00:00:00.641		0xabba	0xcafe	0xbabe	0xcd	Data
6519	20	16:21:52.848	+00:00:00.003		0xbabe	0xcafe	0xabba	0xcd	Data
6520	20	16:21:52.880	+00:00:00.032		0xabba	0xcafe	0xbabe	0xd0	Data
6521	20	16:21:52.883	+00:00:00.003		0xbabe	0xcafe	0xabba	0xd0	Data
6522	20	16:21:53.050	+00:00:00.168		0xabba	0xcafe	0xdead	0xe1	Data
6523	20	16:21:53.053	+00:00:00.003		0xdead	0xcafe	0xabba	0xe1	Data

Figure 4.13: The Daintree Sensor Network Analyzer List View provides a list of packets that have been captured.

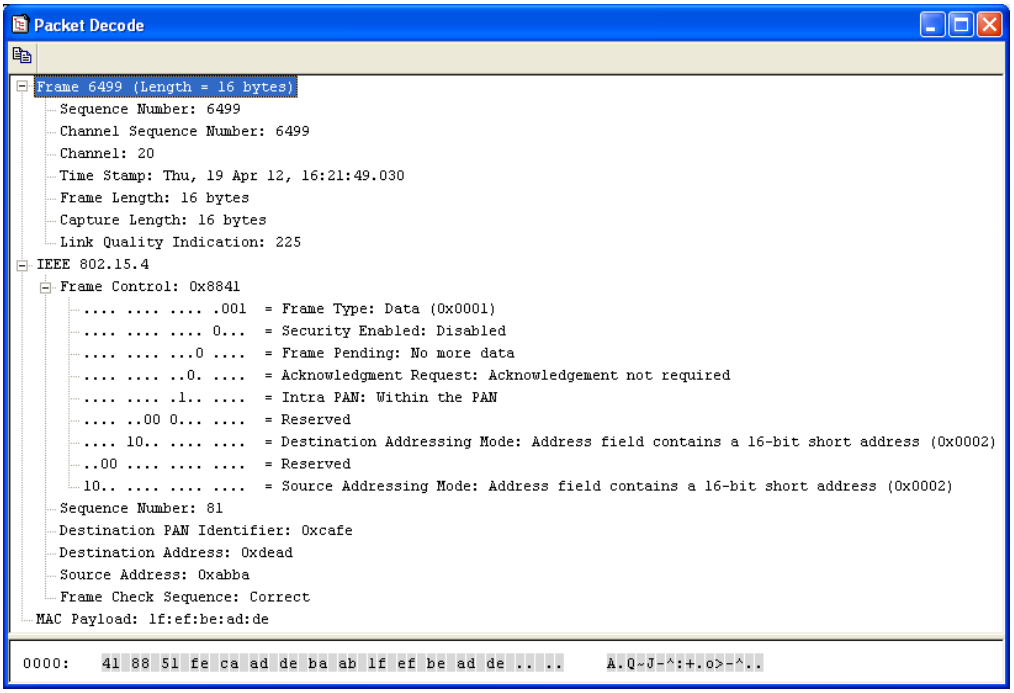


Figure 4.14: The Daintree Sensor Network Analyzer Packet View provides a detailed listing the packet contents and its headers dissected down to IEEE 802.15.4. The WirelessHART headers is not dissected here and is only displayed as part of the payload.

4.2.3 The Contiki Operating System

The Contiki OS[2] is an OS for specific micro controllers used in embedded systems and wireless sensor networks. It has support for the RavenUSB sticks and can easily be programmed onto these. Among other functionalities it supports sniffing of 802.15.4 packets and can identify itself as a network interface in Wireshark. In addition to this it can also be used to send and receive 802.15.4 packets which makes it possible to use the RavenUSB stick as an Access Point for a Network Manager, this will be further described in 7.2.1.1.

4.2.3.1 Using the USB Stick as a Packet Sniffer

We thought about writing our own driver for the USB sticks to enable us to use them as sniffers together with Wireshark and to use them as

Access Points, but soon figured out that would probably be a whole new project that would take a considerable amount of time. We found out about the open source Contiki OS, which has the RavenUSB stick (Jackdaw) platform that can be programmed onto the RavenUSB sticks as an OS. The platform allows the user to use the RavenUSB sticks, amongst others, as sniffers while being considerably more dependable than the Daintree Sensor Network Analyzer as discussed in 4.2.5.

First we downloaded Contiki and reprogrammed the RavenUSB stick through the JTAG interface. After successfully installing Contiki on the USB stick we plugged it into one of our Ubuntu 11.10 test bed machines. The machine instantly recognized the USB stick as a RNDIS Ethernet interface. As described in the previous section a few configurations need to be done before we can enter the Jackdaw Menu and change the OS settings. One of these settings is to set the USB stick in sniffer mode and another to enable it to capture raw packets. After setting these two we need to set the channel to 20 MHz.

4.2.3.2 Using the Contiki OS on the Raven USB Stick

In order to use the Contiki OS on the RavenUSB, or Jackdaw, as the platform intended for the RavenUSB is called, certain steps must be followed. The OS comes with a variety of make targets, 3 of which are designed for ATMEL microprocessors (ATmega3290, ATmega1284 and ATmega1287P). The makefile utilizes the avr-gcc compiler that comes with the AVR Studio programming suite. This in turn produces an object file that has to be written to the EEPROM of the USB-Stick with the use of both AVR Studio and the JTAGICE debugging tool.

After writing the object files, the USB stick identifies itself as a RNDIS device to our Linux host, as well as a debug-COM port that can be used to set environment variables on the USB stick. Examples of these are channel, output power, network-/sniffer-mode.

The debug COM port is available through the `/dev/ttyACM0` with the use of MINICOM and provides a terminal interface for configuration of

Jackdaw (Listing 4.1).

```
********** Jackdaw Menu ******************
*                                          *
* Main Menu:                               *
*  h,?              Print this menu        *
*  m                Print current mode     *
*  s                Set to sniffer mode    *
*  n                Set to network mode    *
*  6                Toggle 6lowpan         *
*  r                Toggle raw mode        *
*  u                Switch to mass-storage *
*                                          *
* Make selection at any time by pressing   *
* your choice on keyboard.                 *
```

Listing 4.1: Jackdaw Configuration Screen

When we run a dmesg (driver message) command on our Linux host we get the output in listing 4.2 (only relevant info is shown).

```
usb 5-2: new full speed USB device using uhci_hcd and
    address 29
usb 5-2: configuration #1 chosen from 1 choice
rndis_host 5-2:1.0: dev can't take 1338 byte packets (max
    1338), adjusting MTU to 1280
usb0: register 'rndis_host' at usb-0000:00:1d.3-2, RNDIS
    device, 02:12:13:14:15:16
cdc_acm 5-2:1.2: ttyACM0: USB ACM device
usb 5-2: New USB device found, idVendor=03eb, idProduct=2021
usb 5-2: New USB device strings: Mfr=1, Product=2,
    SerialNumber=3
usb 5-2: Product: RZRAVEN USB DEMO
usb 5-2: Manufacturer: Atmel
usb 5-2: SerialNumber: 1.0.0
```

Listing 4.2: dmesg output

We then run the command `ip link set usb0 up` and are then able to utilize the `usb0` handle as a NIC as expected.

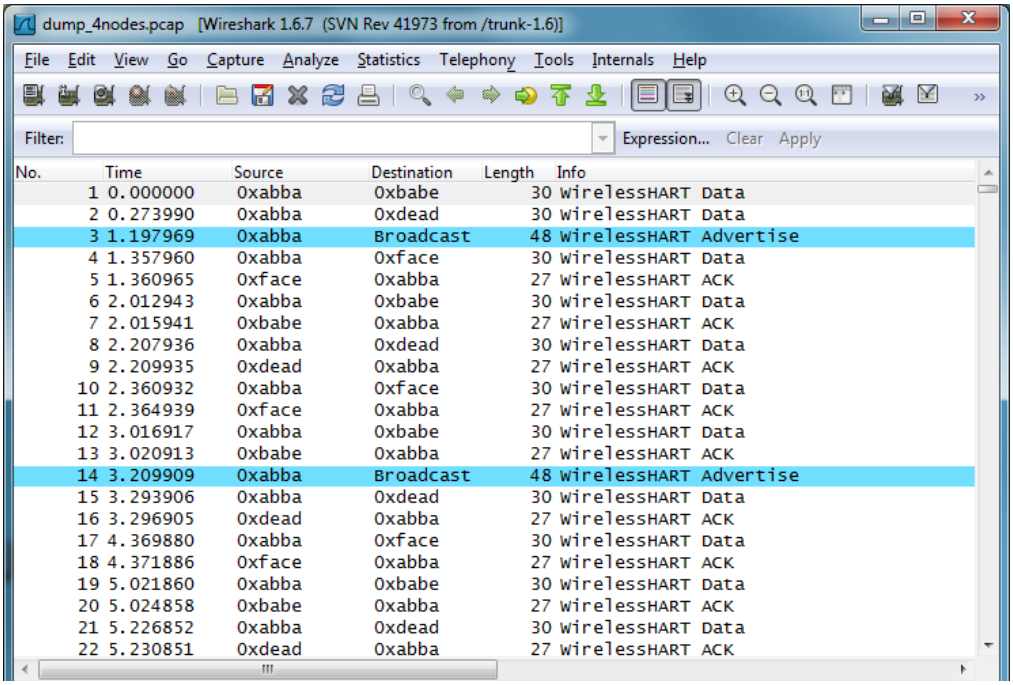


Figure 4.15: A Wireshark Packet Listing with the WirelessHART dissector enabled showing communication between 4 different nodes. Represented by blue color are advertisements that are configured to be sent in fixed intervals.

4.2.4 15dot4-tools

The 15dot4-tools Project[1] supports some of the same functionalities Contiki does, it is however a lot smaller and simpler than the Contiki OS. 15dot4-tools can also be programmed onto the RavenUSB sticks, but runs specific functionality instead of behaving as a complete OS. Although there are several tools only the 802.15.4 Sniffer Tool is supported on the RavenUSB stick. It is, however, possible to adapt the 802.15.4 Raw Packet RX/TX Tools to the Raven USB sticks, more on this in 7.2.1.1.

4.2.5 Wireshark

Wireshark[18] is an open source network protocol analyzer, it is supported in all major operating systems and is very powerful. It offers dissectors for

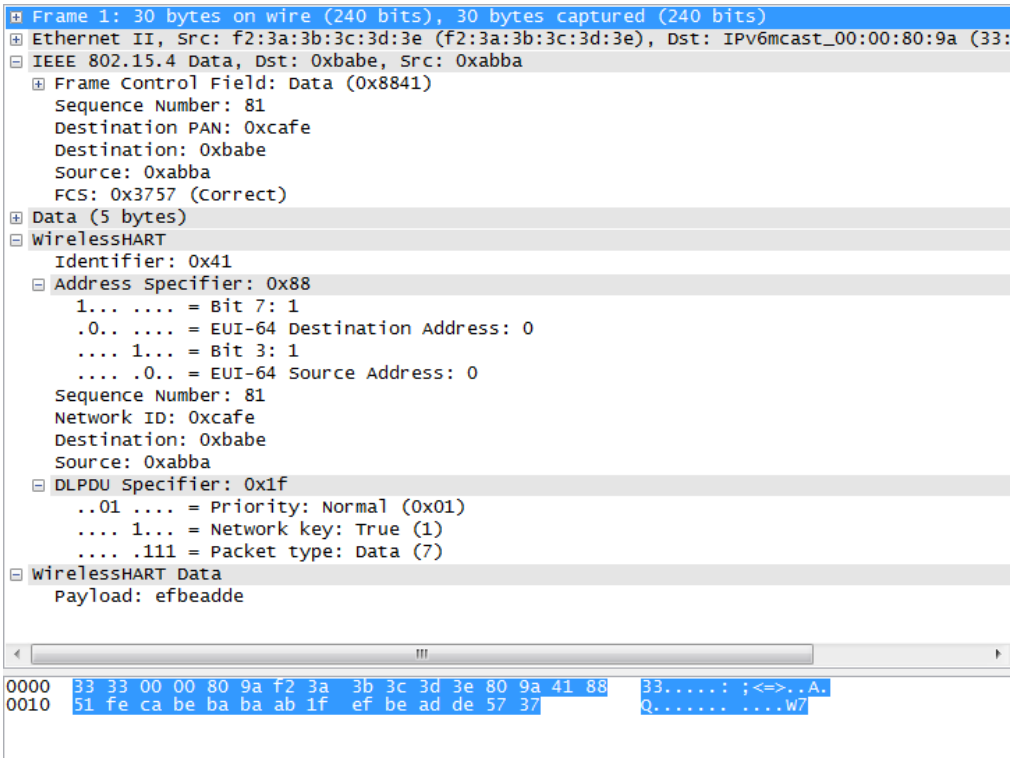


Figure 4.16: The Wireshark Packet Details also offers dissection of the WirelessHART headers

a variety of protocols including IEEE 802.15.4 and ZigBee. In figure 4.15 there is an example of a packet listing of a dump made by Wireshark.

As we managed to get the USB stick to capture raw frames we also wanted to look into using Wireshark as our network sniffer of choice since we had a lot of problems with the Daintree Network Analyzer as described in section 4.2.2.

We were able to simply configure Wireshark to capture packets from an Ethernet device emulated by the USB stick. In the beginning Wireshark recognized the captured packets as IEEE 802.15.4 frames with a ZigBee payload. In order to make it read the packets as raw 802.15.4 frames we needed to disable the decoding of the ZigBee protocol. This means that we can capture raw frames and that the payload of a frame is a WirelessHART packet.

Wireshark in combination with the RavenUSB sticks running the Contiki OS, as described in 4.2.3, was a good solution to our issues with the Daintree Sensor Network Analyzer software. When comparing the two we noticed Daintree had a tendency of dropping packets, while Contiki managed to capture all the packets. We therefore chose to totally abandon Daintree and use Contiki in combination with the RavenUSB sticks and Wireshark as sniffers instead.

Wireshark does not, however, offer a dissector for WirelessHART so out of the box it is only able to decode packets down to the IEEE 802.15.4 header and leave the rest of the packet as raw hexadecimal payload. In order to use Wireshark for debugging WirelessHART packets we have implemented our own dissector for WirelessHART which covers the basic packet types on the link layer. In figure 4.16 there is an example of the available information when examining a specific WirelessHART packet with the WirelessHART dissector enabled. This may be further extended to the network layer as needed. More information about this dissector follows in section 4.2.5.1.

The ability to use Wireshark for basic debugging of packet flow has helped us tremendously in the debugging of the node behavior during sending and receiving since it provides a stable and reliable environment for

packet capture.

4.2.5.1 Dissector for WirelessHART

We examined the option of extending Wireshark and making it parse and show the contents of WirelessHART specific link layer and network layer packets. In this context we have written a simple WirelessHART dissector using the lua[8] scripting language. This works surprisingly well after jumping over some initial hurdles and provides us with the means to debug and examine packet content without having to read raw hexadecimal payloads. Examples of this can be seen in listings 4.3, 4.4 and 4.5.

```
1  IEEE 802.15.4 Data, Dst: Broadcast, Src: 0xabba
2      Frame Control Field: Data (0x8841)
3          .... .... .... .001 = Frame Type: Data (0x0001)
4          .... .... .... 0... = Security Enabled: False
5          .... .... ...0 .... = Frame Pending: False
6          .... .... ..0. .... = Acknowledge Request: False
7          .... .... .1.. .... = Intra-PAN: True
8          .... 10.. .... .... = Destination Addressing Mode:
               Short/16-bit (0x0002)
9          ..00 .... .... .... = Frame Version: 0
10         10.. .... .... .... = Source Addressing Mode: Short
               /16-bit (0x0002)
11     Sequence Number: 100
12     Destination PAN: 0xcafe
13     Destination: 0xffff
14     Source: 0xabba
15     FCS: 0x2ea8 (Correct)
```

Listing 4.3: Example of IEEE802.15.4

```
1  WirelessHART
2      Identifier: 0x41
3      Address Specifier: 0x88
4          1... .... = Bit 7: 1
5          .0.. .... = EUI-64 Destination Address: 0
6          .... 1... = Bit 3: 1
7          .... .0.. = EUI-64 Source Address: 0
```

```
8     Sequence Number: 100
9     Network ID: 0xcafe
10    Destination: 0xffff
11    Source: 0xabba
12    DLPDU Specifier: 0x19
13        ..01 .... = Priority: Normal (0x01)
14        .... 1... = Network key: True (1)
15        .... .001 = Packet type: Advertise (1)
16    Message Integrity Code: 0x0000a82e
```

Listing 4.4: Example of WirelessHART DLPDU

```
1  WirelessHART Advertise
2     Absolute Slot Number: 26
3     Join control: 0x00
4     Channel map size: 64
5     Channel map: 0000100000000000
6     Graph ID: 0x0000
7     Number of superframes: 1
8     Superframe ID: 1
9     Superframe number of slots: 300
10    Number of links: 3
11    Link join slot: 50
12    Link join bits(6:xmit,5-0:chanoffset): 0x00
13    Link join slot: 150
14    Link join bits(6:xmit,5-0:chanoffset): 0x00
15    Link join slot: 250
16    Link join bits(6:xmit,5-0:chanoffset): 0x00
```

Listing 4.5: Example of WirelessHART Advertise DLPDU

At the current state, only support for Data and Advertise packets on the WirelessHART link layer is supported in addition to the native Wireshark support for IEEE 802.15.4. However, the script maybe easily extended to support other packet types both on the link layer and on the network layer, this ability proves extremely useful when debugging packets on these higher layers.

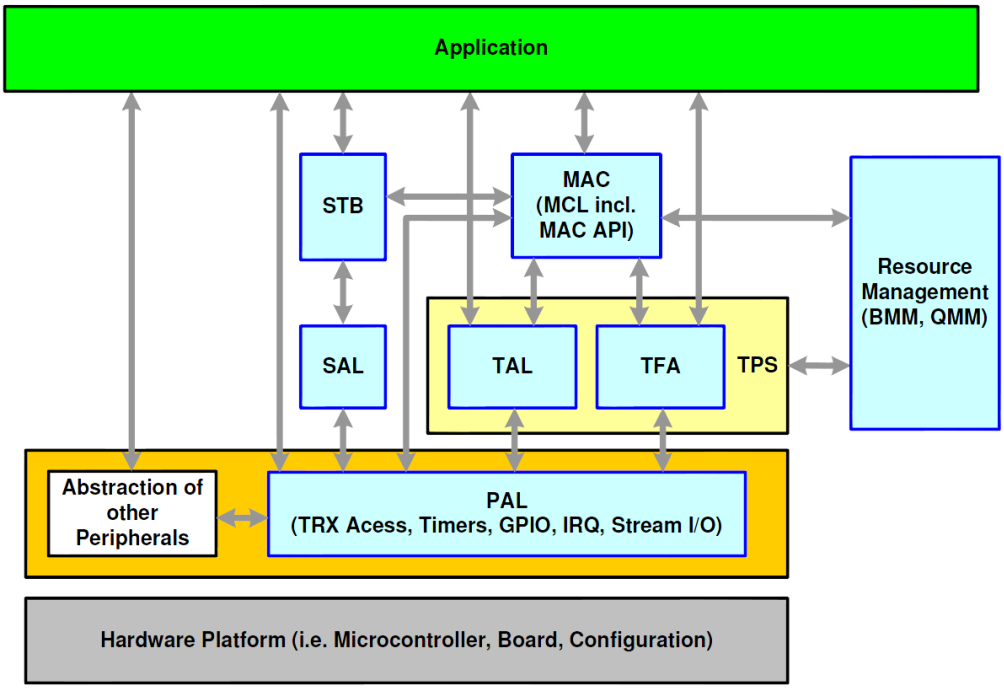


Figure 4.17: AVR2025 Library Architecture - An overview of how the various packages in the AVR2025 library relate to each other (Figure [23, Figure 2.1]).

4.2.6 AVR2025

AVR2025 is a software library developed by Atmel. It supports a full fledged IEEE 802.15.4-2006 implementation and is written to support Atmel's family of micro controllers. The core library consists of three main modules and the original intention of the AVR2025 library is to allow an application to use any layer as desired depending on the required functionality.

In this section we will provide a brief overview over what the original AVR2025 library(Figure 4.17) provides, and how it is organized. The implementation specific details and modifications in relation to this project are described in chapter 5.

4.2.6.1 Platform Abstraction Layer (PAL)

The primary goal of the Platform Abstraction Layer (Figure 4.17) is to separate the functionality of specific micro controllers and provide a common interface for accessing the different models. Proper separation of the PAL from the rest of the software stack is important in order to keep the higher layers indifferent to the specifics of the hardware and allow easy porting to other chipsets.

4.2.6.2 Transceiver Abstraction Layer (TAL)

The Transceiver Abstraction Layer (TAL) seen in figure 4.17 is, as the name suggests, responsible for providing an interface to the transceiver functionality without caring about the specific underlying transceiver. With this important point in mind, as long as we are able to implement our data link plus higher layers and at the same time eliminate or minimize direct access to the PAL and instead use the TAL, we should be able to run the implementation on any micro controller that satisfies the physical requirements defined by the WirelessHART standard [29].

4.2.6.3 MAC Core Layer (MCL)

The MAC Core Layer (MCL) seen in figure 4.17 provides an interface for a IEEE 802.15.4 compliant network to the Network layer. The standard AVR2025 MCL library consists of a series of building blocks providing an event-based execution cycle. We do not delve any deeper into the implementation of the MCL layer since this layer has been partially if not completely rewritten to adapt to the needs of WirelessHART.

4.2.6.4 Resource Management

The AVR library also contains some modules for managing resources, there are two primary data structures for this which are Buffer Management (BMM) and Queue Management (QMM). Queues will most certainly play an important role in our implementation of the data link layer, and will

be used by the TAL in order to store ingress and egress packets. The buffer management provides a convenient interface for managing memory and allocating buffers.

4.3 Chapter Summary

In this chapter we have provided an introduction to the hardware and software available during the project including a discussion of which tools we believe are best suited for the job. This includes both tools for managing hardware, debugging and sniffing packets. We have also provided an introduction to the AVR2025 software library provided by Atmel prior to any modifications in order to give an overview over the original architecture. The changes made to this architecture by Tegelsrud and Frøysadal in [41] will be covered in detail in chapter 5 and our changes are described in chapters 6 and 7.

Chapter 5

Summary of Previous Work

In this chapter we provide a brief introduction to Tegelsrud and Frøysadal's thesis and implementation, a description of each of the changes that were made to the AVR2025 library by them in order to make it compliant to the specification and requirements of WirelessHART and a short outline of our experiences when we first tried to run their code.

5.1 Background

The implementation of WirelessHART that has been performed through the course of this project is based on the thesis "WirelessHART, Gjennomgang og implementering" ("WirelessHART, Review and implementation") [41] by two former students, Håvard Tegelsrud and Jørgen Frøysadal, at the University of Oslo in May 2010. In their thesis the two students reviewed WirelessHART and partially implemented it on Atmel hardware.

Their report covers an overview of the world of wireless sensor networks along with a detailed introduction to the WirelessHART standard. In their conclusion, Tegelsrud and Frøysadal state that they have implemented several key elements in the WirelessHART protocol and that they believe it is possible to fully implement a WirelessHART protocol stack on hardware provided by Atmel even though this hardware was originally intended for use in other wireless sensor networks. In addition they provided a detailed

description of their design and decisions.

Furthermore they have performed a complete implementation of the WirelessHART physical layer on the AVRRZRaven nodes, this includes functionality to send and receive data using the radio. In the addition to this they also designed and partially implemented the link scheduler with the send and receive engines for the MAC sub-layer.

Due to the limited time frame Tegelsrud and Frøysadal only managed to implement a partial Data Link Layer which included both a Logical Link Control (LLC) sub-layer and a Medium Access Control (MAC) sub-layer. Some functionality for DLPDU construction and the TDMA state machine had also been designed but only partially implemented.

The AVR2025 library which is described in section 4.2.6 was used as a starting point for the implementation. According to the report [41] on the code that was previously implemented, three major modifications have been done to this library in order to make it conform to the requirements specified by the WirelessHART specification. An overview of the AVR2025 library with modifications by Tegelsrud and Frøysadal is shown in figure 5.1.

5.2 Decoupled MAC-functions from Hardware

The Atmel chip supports various MAC-layer functionalities in hardware and will automatically perform these functions if the correct hardware registers are set. Since not all of these are compatible with WirelessHART some of this functionality needs to be disabled, and these functions are described in detail in the following sections.

5.2.1 Removed ACK from TAL, Implement it on the Data Link Layer

ACKs are supported on the Transceiver Abstraction Layer (TAL) which is closer to the hardware than the MAC layer, but since this function is

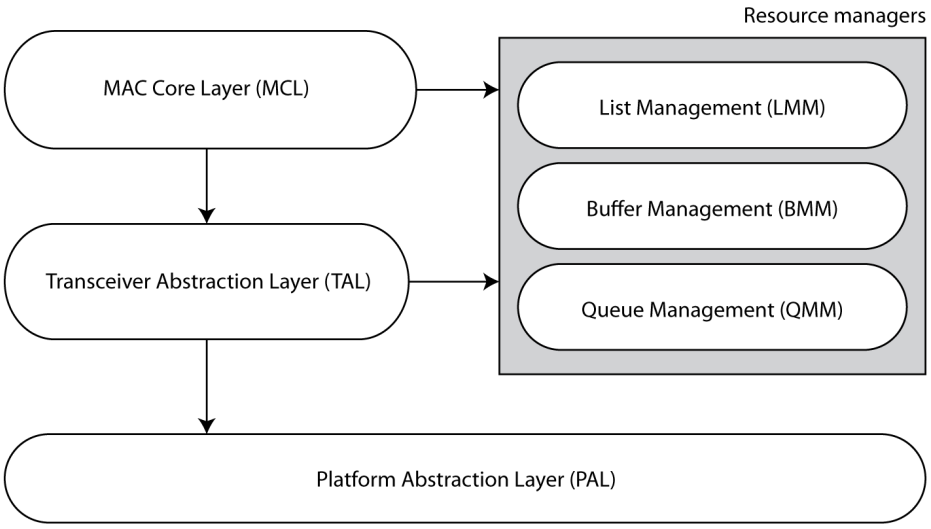


Figure 5.1: AVR2025 Library Overview - A simplified view of the packages from the AVR2025 library used in this implementation in addition to the added List Management resource manager.

not compatible with the WirelessHART protocol the acknowledgment of received packets have been re-implemented on the MAC layer.

5.2.2 Removed Automatic Retransmission from the TAL

Automatic retransmission in WirelessHART is handled by the link scheduler which is responsible for finding a new time slot for sending the packet if no ACK was received as opposed to the automatic retransmission in IEEE 802.15.4 which sends immediately after a fixed timeout.

5.2.3 Deactivated CSMA/CA and Re-implement Clear-Channel-Assessment

WirelessHART only uses Clear Channel Assessment in scenarios where there are shared links, the automatic CCA of outgoing transmissions have

been disabled and instead made available through service primitives. During the initial implementation[41] support for shared links have been discussed but not implemented.

5.3 Implementation of Static Information in the PAN Information Base (PIB)

Some of the fields in the PAN Information Base (PIB) are clearly defined in the WirelessHART standard, but are left as more flexible options in IEEE 802.15.4. In order to simplify the implementation, these fields can be statically set in the PIB. These configuration parameters are explained briefly in appendix B.

For a more detailed reference and explanation of these attributes we refer to the table describing physical layer requirements adopted by WirelessHART [29, 13.3.1.2.2] and the table describing PIB attributes in IEEE 802.15.4 [15, 6.4].

The following list is the the constants from the PIB as they exist in the implementation of Håvard Tegelsrud and Jørgen Frøysadal [41].

- Implement the ISM band selection as a constant (2.4GHz band in IEEE 802.15.4 PHY)
- phyChannelsSupported 11 to 15
- phyMaxFrameDuration is 266
- phySHRDuration 10
- phySymbolsPerOctet 2
- phyTransmitPower should be controllable through local management functions
- phyCCAMode should always be 2 in WirelessHART

5.4 Adapt Service Primitives

The service primitives in this project [41] have been implemented as pure function calls. The exact implementation is vaguely described in the report and mostly refers to the source code.

The data link layer in WirelessHART is very different from the data link layer in IEEE802.15.4 and previous efforts have started on a full re-implementation of this layer. In addition a very basic application layer that communicates directly with the data link layer in order to test the implementation has also been implemented.

As the implementation of Tegelsrud and Frøysadal [41] is only in the first phase of implementation; the WirelessHART protocol stack is not complete, and some of the more significant missing features are:

- Time synchronization
- Routing
- Security
- Configuration messages
- Handling of RX/TX queues
- A functioning TDMA state machine

It is duly noted that the time synchronization and the processing of configuration messages are critical for a functioning WirelessHART network.

5.5 Link Scheduler

The link scheduler is a very important part of the implementation of the WirelessHART protocol, yet the standard is very vague on how the scheduler should be implemented. This has been researched in depth by the previous students and the link scheduler as it exists in the code from Håvard

Tegelsrud and Jørgen Frøysadal [41] has been designed and implemented by them.

While the standard does not explain the implementation of a scheduler in detail, there are a few restrictions that are imposed.

- When a slot has both a packet waiting to be propagated and receive links, propagation of the packet shall have priority over attempting to listen for a new packet.
- Every event that can affect link scheduling shall result in the link schedule being re-assessed

The first rule we do not need to worry about at this point as the network does not currently support multi-hop or routing, thus a receive link will never be halted in favor of a packet that requires propagation. This does not, however, mean that we disregard multi-hop in the future and this design will not prevent from extending to support multi-hop. The second rule is an important one and we will have to make sure that any modification to the superframes or neighbor tables, in addition to reception and transmission of a packet, results in an execution of the link scheduler.

Energy Consumption

Low energy consumption and by extension, a very long battery life, is right in the heart of the wireless sensor networking world. The link scheduler has an important function in this as it is the scheduler that decides when to go to sleep and when the node is required to wake up again. While the requirements for when to serve a slot are strictly defined by the WirelessHART standard, we need to consider the possibilities of optimizing the scheduler in a way that allows for maximum sleep time between slots. This means that link selection has to happen as fast and with as low complexity as possible.

Efficiency

Since the link scheduler usually has to run within the execution of a time slot the efficiency and performance requirements are quite high.

The selection of the next link to schedule needs to be fast enough to finish within the given time constraints. While this currently has been tested with relatively small networks and a small number of links the performance of the scheduling algorithm and accompanying data structures needs to be evaluated when there are multiple super frames with a high number of links in play.

Fairness and Starvation

It is the responsibility of the link scheduler to ensure that all the links get treated fairly and that packets with high priority get better service without compromising the integrity of the network by starvation. As the nodes operate with a very limited amount of memory buffers for storing packets, the scheduler has to ensure that starvation will not happen due to lots of low-priority packets being delayed in favor of high-priority packets. A result of this includes high priority incoming packets being dropped due to there not being any available buffers. As far as the standard allows, the scheduler needs some way of controlling this such as dropping or discarding unimportant packets.

The functionality of the link scheduler that was implemented by the former students can be described in two sentences; "Retrieve the first link to be processed from each active superframe in a list ordered by the ASN number of the link. Select and process the link with the highest priority from the list based on the criterias defined by the WirelessHART standard."

5.6 Running the Code

In the beginning when we tried to run the source code provided by the previous students we ran into some issues. Some of these issues were simply due to incomplete timing mechanisms and were to be expected but some of the issues were also caused by memory leaks and incorrect use of local stack variables in the implementation of the list management

functions. At the time we were not aware of the fact that some parts of the implementation were not finished or tested.

We did spend a lot of time debugging these problems, since not only were we unfamiliar with the code base at the time, but the problems we encountered were unexpected as we believe the code was supposed to work as it was. We therefore assumed we had done something wrong while setting up the environment. Once we got over these hurdles and took a closer look at the implementation, these problems could be fixed quite easily.

5.7 Chapter Summary

In this chapter, we have in addition to a short summary of the report Tegelsrud and Frøysadal has written provided an overview of what work has previously performed by them on this project. We have also provided a detailed view of which changes have been made to the original AVR2025 library prior to the start of our project, and this modified library provides a baseline for the continuation of the project during our design phase. And the end of the chapter we have provided a short introduction to our experiences when running their code for the first time and our efforts to make the initial implementation function correctly.

Chapter 6

Design

Throughout this chapter we provide an overview of the key areas of which we decided to focus on, based on the work previously performed and described in chapter 5. After we have defined the main focus areas we then move on to defining a set of functional and non-functional requirements which serves as a metric for successful completion when implementing and evaluating the project. In addition this chapter will provide a basic design of new features both on the WirelessHART Field Devices and a rudimentary WirelessHART Gateway. The key focus areas along with the requirements will form the basis of the design and clarify which parts of the WirelessHART stack need to be implemented first.

6.1 Key Focus Areas

We identified several key areas that require further development on the data link layer of the WirelessHART stack. Each of these key areas are further segmented and functional requirements are defined in section 6.2.

TDMA State machine

Finish the implementation of the TDMA state machine designed in [41] and at the same time take the opportunity to view and analyze the suggested design.

Time Synchronization

Basic time synchronization is crucial for a functioning WirelessHART network. In order to set up a basic testing environment all we need is the existing functionality of a node to be able to send a packet. This functionality already existed in the code base and allows the node to synchronize itself every time a packet is received.

Local Management Service Primitives

Since these primitives include functionality for letting nodes join the network, for advertising and for controlling superframes, these are crucial for the implementation of a Network Manager, and to set up a simple test network with multiple nodes. In short, this is part of the groundwork for nodes being able to dynamically join and leave the WirelessHART network.

Implementing a Gateway/Network Manager

In order to have a fully functional WirelessHART network a rudimentary Network Manager and Gateway needs to be implemented in order to be able to manage and monitor the network. In the first round, the Gateway primarily needs the ability to create and send superframes to the nodes based on the topology of the network.

6.2 Functional Requirements

Functional requirements are requirements that are easily measured based on the completeness and accuracy of their implementation. Table 6.1 provides an overview of the functional requirements we have based our work on, each of the requirements have a priority ranging from LOW to HIGH in addition to an ID which is used for referencing specific requirements. The priorities are set with the primary goal of finishing the MAC layer functionality. Each of the requirements defined in the table is then described in more detail in the following sections.

ID	Requirement	Priority
FR01	A Working XMIT and RECV Engine on the Link Layer	HIGH
FR02	Service Primitives for Local Management	LOW
FR03	Multiple RX and TX Queues	MEDIUM
FR04	Interface Between Computer and Sensor Nodes	HIGH
FR05	Time Synchronization	HIGH
FR06	Network Manager	HIGH

Table 6.1: Functional Requirements

6.2.1 Requirement Specification

In this subsection we provide a description of each of the requirements specified in table 6.1.

6.2.1.1 A Working XMIT and RECV Engine on the Link Layer (FR01)

Finish the implementation of the TDMA state machine in context of the XMIT and RECV engine in order to reliably be able to send and receive packets. Until this is finished, we will not be able to properly implement and test timing mechanisms or communication with a Network Manager.

6.2.1.2 Service Primitives for Local Management (FR02)

Implement the necessary service primitives for local management for use by the network layer application to manage the data link layer structures. These primitives need to be prioritized, designed and implemented in the MAC API.

6.2.1.3 Multiple RX and TX Queues (FR03)

The previous implementation contained a single global TX queue in addition to each neighbor having a linked list to the packets it owns in this TX queue. We reviewed and redesigned this approach a little and decided to rewrite this so that instead of having a global TX queue each neighbor

will have its queue of packets waiting to be sent. This simplifies the implementation quite a bit as there will only be a single point of entry in the queues. There is currently no reason as far as we can see to keep a global TX queue.

6.2.1.4 Interface Between Computer and Sensor Nodes (FR04)

In order to start implementing a basic Network Manager we need to design, implement and test a way to communicate between a computer and the sensor nodes. Should this not be possible within the given time frame we will implement basic Network Manager functionality on a sensor node for testing purposes and outline the required changes for a full-blown Network Manager in the future work chapter.

6.2.1.5 Time Synchronization (FR05)

In order for nodes to be able to communicate on the same network they need to have the same definition of the system time. This means that in order to test basic send/receive functionality and any communication with the Network Manager we first need to implement basic primitives for synchronizing time keeping between multiple nodes. A more detailed design of time synchronization methods are described in section 3.6.

6.2.1.6 Network Manager (FR06)

A rudimentary Network Manager is required in order to have a functional and maintainable WirelessHART network. While the communication between the Network Manager and the WirelessHART network relies upon the completion of FR04 (Table 6.1) we will still be able to design and implement a basic interface and core architecture for the Network Manager.

6.3 Non-functional Requirements

Non-functional requirements are requirements that are not so easily measured but define general characteristics on the project, its code base and

the provided documentation. Table 6.2 provides an overview of the non-functional requirements related to the project, each of the requirements have a priority ranging from LOW to HIGH in addition to an ID which is used for referencing specific requirements. The priorities are set with the primary goal of providing a well documented and easily maintainable code base for future work. Each of the requirements defined in the table is then described in more detail in the following sections.

ID	Requirement	Priority
NFR01	Processing of a Specific Slot Within Timing Limitations	HIGH
NFR02	Proper Documentation of API	HIGH
NFR03	Documentation of Non-API Functions	MEDIUM
NFR04	Low Power Consumption, Minimal use of Radio	LOW
NFR05	Separation of Layers	MEDIUM

Table 6.2: Non-functional Requirements

6.3.1 Requirement Specification

In this subsection we provide a description of each of the requirements specified in table 6.2.

6.3.1.1 Processing of a Specific Slot Within Timing Limitations (NFR01)

Being able to process a time slot within the imposed timing limitations is crucial for a functioning WirelessHART network. We need to take measurements and verify that a node is able to perform its duties within the given time constraints and if not, take steps to ensure it does.

6.3.1.2 Proper Documentation of API (NFR02)

Proper documentation of the API is crucial in order to write semantically correct, layered code. The API documentation needs to provide a clear overview of which functionality is exposed to the higher layers as service primitives and explain their behavior in great detail. We need to assume

that the only piece of information a programmer will use to write code on top of this library is the API documentation and this should form the cornerstone in the documentation effort along with an overview of the library architecture.

6.3.1.3 Documentation of Non-API Functions (NFR03)

While the documentation of non-API functions does not matter to the application layer programmer, we believe proper documentation and reference is very important when the project needs to be transferred to new students. We spent a lot of time reading and getting to know the code in order to gain a complete picture of what happens, and we should do what we can to minimize the effort needed in the future. The term "documentation of non-API function" will also include writing easy to read and properly commented code.

6.3.1.4 Low Power Consumption, Minimal use of Radio (NFR04)

Reliability and low power consumption is the whole point of WirelessHART. While this does not directly affect the project at this stage, we will examine the possibility of saving power whenever we can. However, the focus will be on completing the tasks within the given timing limitations and power optimization will be a secondary if not a tertiary task.

6.3.1.5 Separation of Layers (NFR05)

In order to prepare for the future we need to provide a clear separation of concerns between the implemented layers and the layers which we will design and implement over the course of this project. This is especially the case in terms of the boundary between the transceiver abstraction layer (TAL) and the link layer as this will be the primary problem area when moving to a newer hardware platform in the future.

6.4 Quality Assurance

In this section we provide an overview of the tools and methods we use to assure that the result is easy to maintain and extend in the future. This includes tools for source code and configuration management as well as tools for generating documentation and code analysis.

6.4.1 Configuration Management

The configuration of the network should mostly be handled by the Network Manager. During development there will be an inherent need to separate the behavior of different sets of nodes. This includes for example programming the nodes with nicknames and network IDs. In order to make this easy to control we will implement a set of macros and function calls to make this easier during the development process. Once the implementation is complete all the configuration aspects will be handled by the Network Manager and these functions and macros will no longer be required. In terms of configuration management for the Network Manager and Virtual Gateway we need to design and implement an interface for managing this within the Network Manager.

6.4.2 Code Inspection

Inspecting the code written for the nodes themselves is a difficult task, here we mostly have to rely on our ability to follow the coding standards already provided by the AVR2025 library.

For the Network Manager and Virtual Gateway which is implemented in Java using tools like Lint4j[7] to analyze the code and check for error conditions and other issues can be very useful. The primary purpose of this tool is to provide the programmer with an edge on code review and eliminate common problem scenarios that are easily detected by scripted tools.

6.4.3 Source Code Management

For maintaining flexibility and control over the code base Subversion will be used. Subversion is a centralized system for revision and version control distributed under a free Apache License. We utilize our Subversion repository hosted by the University of Oslo. Since Subversion works nicely with any set of files that are stored in plain text we can also use the same approach to version control and manage the project report.

6.4.4 Coding Standards

A consistent code base is a requirement for easily maintainable code and to since large parts of the library is already provided, we adhere to the same coding standards and styles used in the AVR2025 library for the programming on the nodes.

For Java coding standards we will use the formatting rules provided by default in Eclipse. This allows us to use the auto formatting rules without much work. We will focus on creating a modular design of the Network Manager which is easily maintained and extended in the future.

6.4.5 Documentation

For providing proper documentation, a hand-written API reference in addition to consistent use of documentation tags for generating reference documentation using Doxygen will be used. Towards the end of the project we will generate and package a complete source code reference. For documentation of the Network Manager and Virtual Gateway we will resort to using JavaDoc as the primary source of documentation.

6.4.6 Testing and Verification

For testing and verification of functionality of communication aspects we will depend on using external wireless packet sniffers in order to confirm that the implementation works as expected. The reason behind this is that while we are able to debug the nodes directly on the hardware using

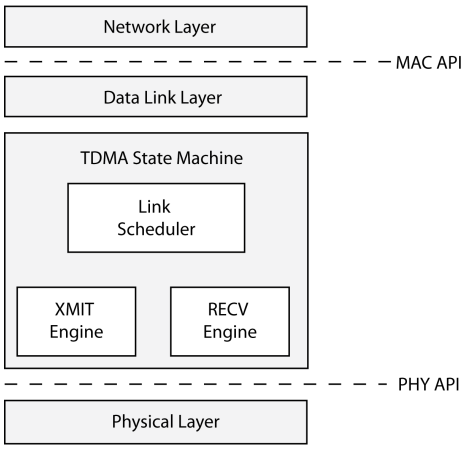


Figure 6.1: Link Layer Architecture

JTAG, this is not a good approach for confirming functionality that have any dependency on timing or other nodes on the network. The debugger will slow down the clock and make it impossible to have any real-time scenario while running the debugger.

6.5 WirelessHART Field Devices

In this section we provide a description of the design we are working towards for completing the implementation of the Data Link Layer on the WirelessHART Field Devices. As displayed in figure 6.1 the main components are the TDMA State Machine, the XMIT and RECV engines and the Link Scheduler. In addition we provide an explanation of the basic design of the Network Abstraction Layer (NAL) as the next step in the protocol staircase.

6.5.1 TX and RX queues

When implementing the queues for transmitting (TX) and receiving (RX) packets there are several key factors that require consideration. All of these actions happen inside time slots so they all have strict timing requirements.

The focus when choosing a data structure for storing these packets should be on speed and memory usage.

For the RX queue the following features are required:

- Get the next packet that has been received and pass it further up the stack
- Store up to N packets

For the TX queue the following features are required:

- Check if the TX link that is being processed has any packets to be transmitted
- If an ACK is not received for the packet, add it back to the TX queue for that node

Based on this information we have designed a basic architecture for packet management which consists of a single RX queue and multiple TX queues (one TX queue per neighbor). We believe that this makes a simple approach to the transmit queuing without losing any functionality compared to the previous implementation. An illustration of this architecture can be seen in figure 7.7.

6.5.2 TDMA State Machine

A TDMA state machine described in section 3.3.2 has been designed and partially implemented by Tegelsrud and Frøysadal, in this section we provide a brief introduction to which parts of the state machine have been implemented and which parts still have to be written in order to complete the design.

While the basic scaffolding for making the TDMA state machine function had already been implemented such as the `LISTEN`, `TALK` and `WAITFORACK` states, it had not been tied together using a common interface for transitioning between states. Since the design of the TDMA state machine is specified fairly detailed in the WirelessHART standard we proceeded

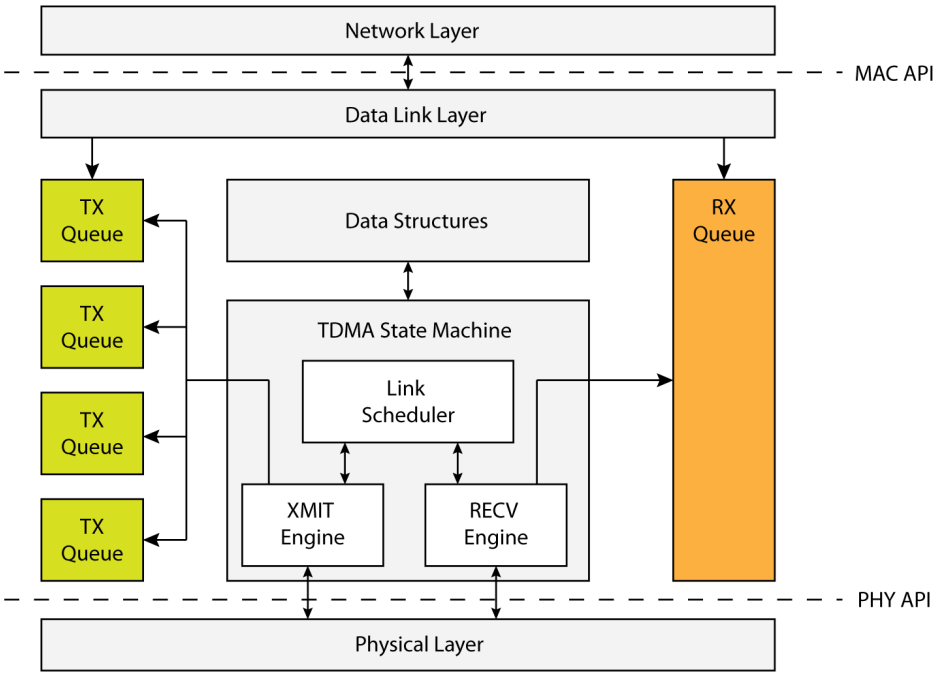


Figure 6.2: Link Layer Architecture, TX and RX Queues - Each neighbor has its own TX queue while there is a global shared RX queue for incoming packets, this differs slightly from the previous design where there were both a global TX and RX queue and the neighbors were responsible for keeping track of their own packets in the queue.

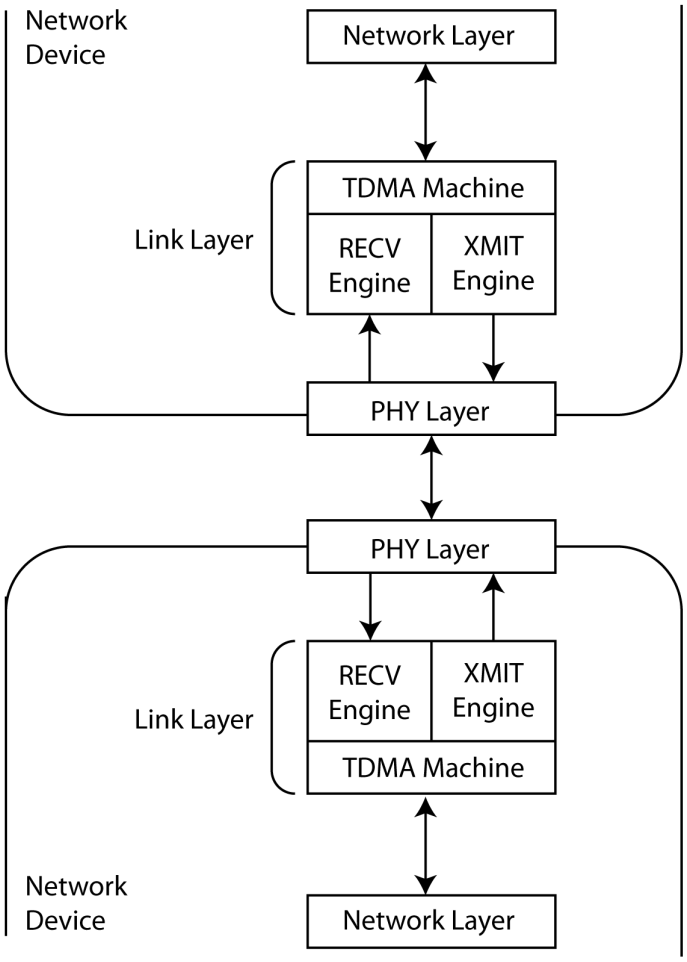


Figure 6.3: WirelessHART MAC Components

to implement function calls and state transitions in the state machine as described in section 7.1.2.

For a better overview of how the state machine and the two engines operate and relate together refer to figure 6.3.

6.5.3 XMIT Engine

The XMIT engine as described by the WirelessHART standard [29] is responsible for providing the functionality to communicate with the physical layer and send a packet. As we can see in figure 6.4 only two execution

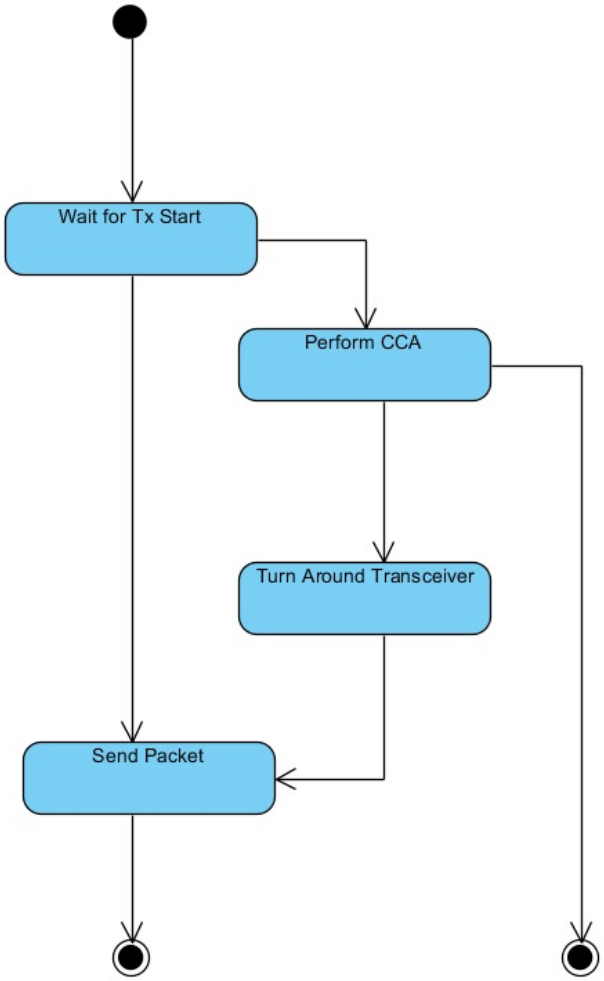


Figure 6.4: XMIT State Machine

paths are possible, with or without Clear-Channel-Assessment (CCA). In our project we will disregard the CCA execution path since that is functionality which is intended for shared slots. We currently do not support shared slots in our implementation so as a result of this, we will focus solely on direct transmission with no CCA.

The XMIT engine will need to provide the following capabilities

- Given the proper input, create a frame to be transmitted on the MAC layer
- Be able to adhere to the timing requirements of the WirelessHART standard in respect to slot timing
- Transmit the frame using the MAC API

6.5.3.1 Frame Creation

During frame creation phase, the XMIT engine will perform the final steps in building the MAC frame before it can be transmitted. This primarily includes building the frame header in the correct buffer and returning a pointer to the buffer. I addition the buffer will be 1 byte longer than the actual frame so that the leading byte in the buffer can contain the total frame length.

6.5.3.2 Frame Transmission

Most of the functionality in this phase is already provided by the MAC API through the `data_request()` function. All the XMIT function really has to do is to wait until it is time to send the frame as specified by the slot timing diagram (Figure 6.5) and call `data_request()` with the proper packet buffer.

6.5.3.3 Timing Requirements

Considering the slot timing diagram in figure 6.5 and due to the fact that we are not implementing shared slots at this point, we can ignore some of the properties in the diagram.

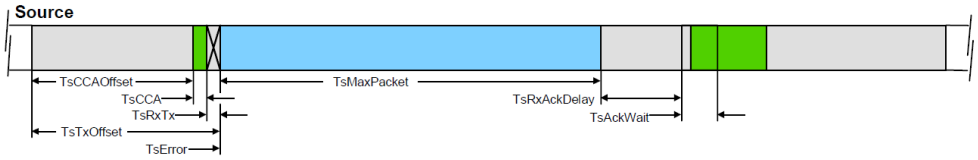


Figure 6.5: Timing diagram for sender

The most important parts for us during slot timing are the following

TxTxOffset which describes how long to wait from the start of the slot before sending the packet

TsMaxPacket which describes the maximum amount of time we are allowed to spend sending the packet

TsRxAckDelay which describes how much time we have to flip the radio and start listening for an ACK

TsAckWait which describes how long a device shall wait for an ACK before declaring the transmission as failed and re-scheduling the packet for a later time

6.5.4 RECV Engine

The RECV engine as described by the WirelessHART standard [29] is responsible for providing the functionality to communicate with the physical layer and receive packets. The act of actually receiving the packet will need to be handled in the interrupt handler but the RECV engine should provide the capability to start and stop listening for packets within the timing requirements provided by figure 6.7. The RECV state machine is illustrated in figure 6.6.

The RECV engine will need to provide the following capabilities:

- Wait for a frame to arrive in the time frame provided by the slot timing

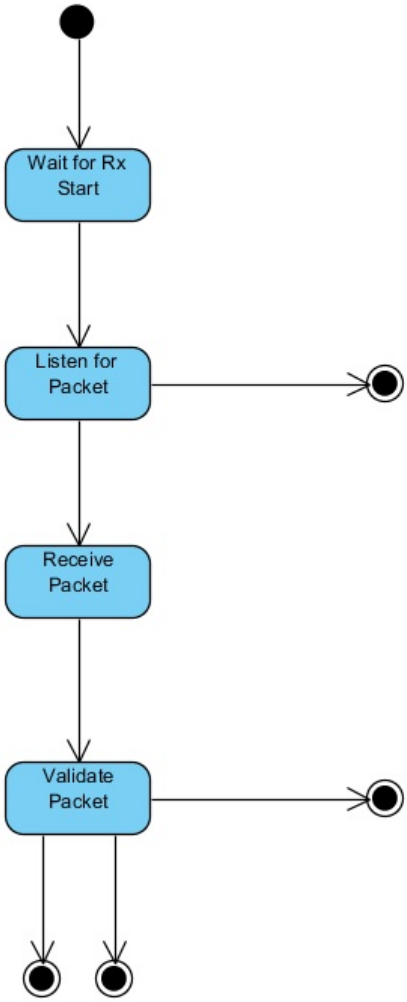


Figure 6.6: RECV State Machine

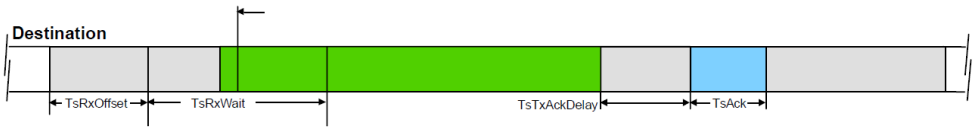


Figure 6.7: Timing diagram for receiver

- Time out if no packet has been received during the wait period
- Receive a packet and add the packet to a receive queue for processing

6.5.4.1 Wait for a Packet

This is a fairly simple procedure which consists of two main parts. First the time at which we should stop listening for a packet is calculated and then a timer is started to call a function to notify the host that no packet was received. If a packet is not received before the timer times out the host will be notified of this, but if a packet is received the timer will be canceled.

6.5.4.2 Time Out if no Packet is Received

If no packet is received before the timer expires, this function is called and is responsible for transitioning the TDMA state machine back to `IDLE` state.

6.5.4.3 Timing Requirements

Considering the slot timing diagram in figure 6.7 and due to the fact that we are not implementing shared slots at this point, we can ignore some of the properties in the diagram.

The most important parts for us during slot timing are the following

TxRxOffset which describes how long to wait from the start of the slot before starting to listen

TsRxWait which when and how long to wait for a received packet

TsTxAckDelay which describes how much time we have to flip the radio to send an ACK

TsAck which describes how much time we have to send the ACK

6.5.5 Design of the Network Abstraction Layer (NAL)

In order to progress further up the stack we need to declare an abstraction between the network layer and the higher layer. The abstraction will be similar to the already implemented interface between the physical layer and the link layer. As on the link layer, this interface will be implemented using pure function calls and parameter juggling.

The WirelessHART specification describes 3 types of `TRANSMIT.request` service primitives. As in the link layer we will join these into a single function call with the required parameters. The `TRANSMIT.request` service primitive is used by the application layer to send a packet to one or more devices in the network, `TRANSMIT.indicate` is used to pass received packets from the network layer and up to the higher layers, `TRANSMIT.confirm` is used to pass the result of a `TRANSMIT.request` up to the application layer and `TRANSMIT.response` is used to notify the application layer that a packet with response data has been received.

The API for the `TRANSMIT` service primitive is shown by listing 6.1. The `TRANSMIT.response` service primitive is executed by the application layer to notify the network layer that the packet has been received along with the response data. When a packet is received by the network layer the `TRANSMIT .indicate` primitive should be invoked and the packet should be transferred to the higher layers. The `TRANSMIT.confirm` service primitive is used to pass the result of a `TRANSMIT.request` back to the application layer.

```
1 void nal_transmit_request(
2     uint8_t handle,
3     uint8_t dst_type,
4     uint64_t dst,
5     uint8_t priority,
6     uint8_t transport_type,
7     uint8_t *payload);
```

```
8
9  void nal_transmit_indicate(
10     uint8_t handle,
11     uint64_t src,
12     uint8_t priority,
13     uint8_t transport_type,
14     uint8_t *payload);
15
16 void nal_transmit_response(
17     uint8_t handle,
18     uint8_t *payload);
19
20 void nal_transmit_confirm(
21     uint8_t handle,
22     uint8_t status,
23     uint8_t *payload);
```

Listing 6.1: Network Layer Transmit Service Primitive

The API for the FLUSH service primitive is used by the transport layer to ask the network layer to delete a packet from the queue. The FLUSH.request service primitive is used to delete a packet identified by the packet handle, after which the FLUSH.confirm primitive is used to notify the caller that the packet was successfully deleted. The API is shown by listing 6.2.

```
1  void nal_flush_request(
2      uint8_t handle);
3
4  void nal_flush_confirm(
5      uint8_t handle,
6      uint8_t status);
```

Listing 6.2: Network Layer Flush Service Primitive

The API for the LOCAL_MANAGEMENT service primitive is shown by listing 6.3 and is used to invoke local management commands on the device.

```
1  void nal_manage_request(
2      uint8_t service,
3      uint8_t *data);
4
```

```
5  void nal_manage_confirm(
6      uint8_t service,
7      uint8_t status,
8      uint8_t *data);
9
10 void nal_manage_indicate(
11     uint8_t service,
12     uint8_t status,
13     uint8_t *data);
```

Listing 6.3: Network Layer Local Management Service Primitive

6.6 WirelessHART Gateway

As discussed in section the WirelessHART Gateway consists of several building blocks and can be further segmented into a Virtual Gateway, Access Points and Network Manager. A block diagram of the Gateway showing all the components it consists of can be seen in figure 6.8. In order to have a fully functional WirelessHART network the Gateway needs to provide the following set of requirements

1. One or more Access Points
2. A Virtual Gateway
3. A connection to the Network Manager
4. One or more host interfaces
5. Time synchronization
6. Buffering and local storage
7. Support for WirelessHART adapters
8. Backwards compatibility with legacy applications

In our design of a Gateway we focus only on the first 4 items, striving to design an easily extendible and well documented prototype application.

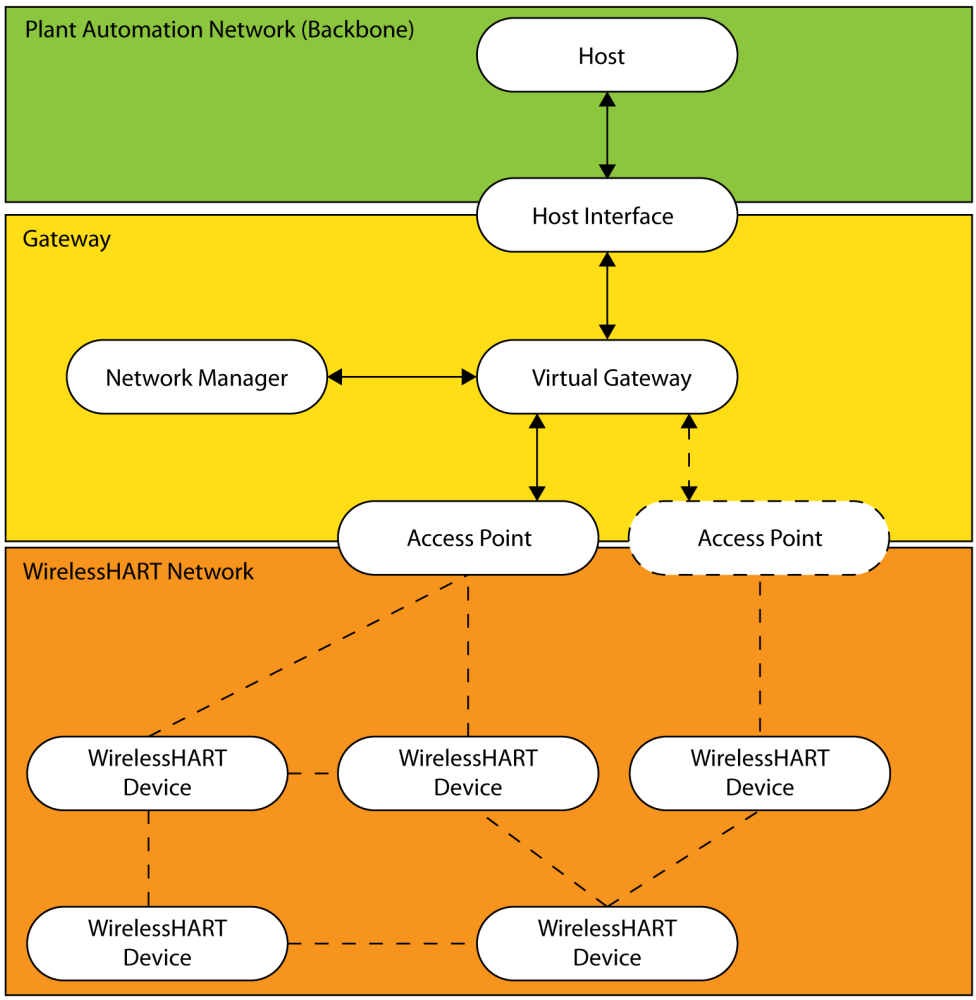


Figure 6.8: Gateway Block Diagram - Showing the relationship between the components of a WirelessHART Network including an overview of the internal building blocks of the WirelessHART Gateway its interfaces to the wireless network through the Access Points and the interface to the Plant Automation Network.

6.6.1 Virtual Gateway

The Virtual Gateway is a stand-alone class which runs in a separate thread within the Gateway. The primary responsibility of the Virtual Gateway is to act as a sink for network traffic from multiple Access Points and host applications (Figure 6.9). The Virtual Gateway is responsible for deciding which traffic needs to be passed to the Network Manager through Command objects. Command objects are further discussed in section 6.6.6.

6.6.2 Network Manager

The Network Manager is as a stand-alone class which runs in a separate thread within the Gateway. It contains all the necessary data structures to manage and modify the routing tables, neighbors, statistics and so forth, including the building of the superframe. The Network Managers primary responsibility will be to respond to commands and requests passed to it from the Virtual Gateway, these commands are further discussed in section 6.6.6 and the interface for communication between the Network Manager and Virtual Gateway is discussed in section 6.6.4.

6.6.3 Access Points

As described earlier each WirelessHART network only has one Gateway and the Gateway must have one or more Access Points. Although the Gateway has a well known address the Access Points do not share this address. Each Access Point has its own unique ID and nickname and if they receive a packet whose destination is the Gateway/Network Manager it relays this on to the gateway. Having several Access Points enables the Gateway to communicate more frequently with the network and also makes it more redundant to Access Point failures.

The way we designed the Gateway and Access Points is to have one Gateway which initiates the Virtual Gateway. The Virtual Gateway then initiates as many Access Points as it wishes. These Access Points now run almost as normal network devices (nodes) with their own nickname, list

of links and neighbors and so on. All the Access Points share a TX-queue which contains packets the Gateway wishes to send. They also share an RX-event queue which is where the received packets for the Gateway are stored by each Access Point. The Virtual Gateway is polling this event queue for any incoming packets periodically and processes whatever it has received.

As we are still in the initial stages of designing a fully functional WirelessHART network the Access Points will be sharing neighbors and links, so all the Access Points will have the same neighbor and link lists.

6.6.4 Communication Between Virtual Gateway and Network Manager

The communication between these logical units can be one of several. They can run on separate geographical locations, thus having being connected through i.e. some form of IP-Based VPN, or be represented as different classes in the same application. In the initial design, communication between the Gateway and the Network Manager will be through command-passing using an event queue. The two entities should execute in different threads allowing them to work independently of each other except for when communication is required. The Gateway will be responsible for passing any relevant commands or messages that are received on the Access Points through to the Network Manager.

6.6.5 Communication Between Virtual Gateway and WirelessHART Network

As with communication between a Gateway and a Network Manager, these can also be separated geographically or logically. Communication is realized through an Access Point that serves as the interface between the production backbone and the WirelessHART device network. In our initial design we will use a single Access Point which executes in its own thread and is responsible for receiving and transmitting packets that are bound

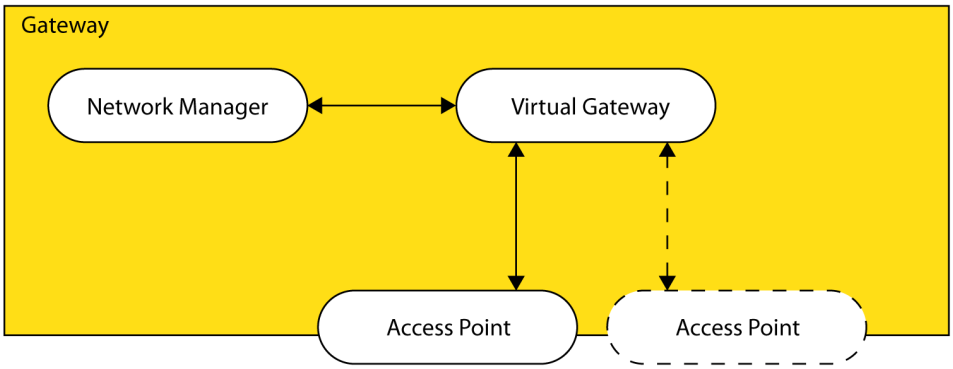


Figure 6.9: Communication Between Gateway Components - Giving an overview of the internal structure of the components in the WirelessHART gateway and the interface to the wireless network. The Virtual Gateway may have several Access Points.

for or coming from the Gateway.

6.6.6 HART Commands Interface

HART Commands will be encapsulated as command objects where each object contains a command code and the necessary arguments to complete the command or mitigate the response. As packets arrive on the Access Point they will be processed by the Virtual Gateway which creates command objects should that be required. The information inside the command object may then be used to invoke a function call to perform the command and return the response. The outline of how this procedure is accomplished will be more thoroughly discussed in section 7.2.2.2.

6.7 Chapter Summary

In this chapter we have provided the reader with an overview of the key focus areas during this project, explained the functional and non-functional requirements associated with this project in addition to basic design proposals for the various requirements. We have outlined an approach for

implementing critical features of WirelessHART Field Devices in order to create a fully functional link layer including the TDMA state machine which is already partially implemented by Tegelsrud and Frøysadal. In addition, we have outlined an approach for implementing the core behavior and functionality of the WirelessHART Gateway.

Chapter 7

Implementation

This chapter includes a detailed description of how the features described in chapter 6 have been implemented, the choices that were made during the implementation, how they differ from the design and the reasons why. First we provide an overview of the implementation efforts one the WirelessHART Field Devices and the continuation of Tegelsrud and Frøysadal's implementation. Afterwards we move on to describing our implementation of the WirelessHART Gateway including its subcomponents. The code written in Java is documented with JavaDoc[34], and the C code is documented with Doxygen [44].

7.1 WirelessHART Field Devices

In this section we provide a detailed description to the various changes made to the implementation of the WirelessHART Field Devices. this includes the implementation of Field Devices' MAC layer API and adjacent support functions including the link scheduler, time synchronization and DLPDU construction.

7.1.1 TX and RX Queues

In order to meet FR03 - Multiple RX and TX Queues (table 6.1) described in section 6.2 we needed to modify Tegelsrud and Frøysadal's implementa-

tion of transmit queues for the neighbors. Instead of maintaining one global transmit queue and having each of the neighbor data structures maintain a linked list to the packets it owns in this queue, we have modified it so that each neighbor data structure maintains its own transmit queue using a proper queue structure. We believe that this makes a simpler approach to the transmit queuing without losing any functionality simply because there are less pointers to keep track of and less methods which the queues can be accessed through. While this modification does not make a serious impact on the execution time because the complexity of accessing the packets remains the same, good encapsulation of functionality makes the code easier to maintain and extend in the future.

When a node transmits a packet, the packet will simply be moved to another queue while it waits for acknowledgment, if required. If no acknowledgment is required, the packet buffer will be freed after a successful transmission. This requires some thought when it comes to broadcast packets as there is no longer a global transmit queue. We solve this by simply creating a fake neighbor that owns the broadcast address so that we have a place to schedule broadcast packets. This neighbor has the nickname 0xFFFF which is defined as the broadcast address in the WirelessHART standard.

7.1.2 TDMA State Machine

The implementation of the TDMA State Machine consists of a set of functions which all end with a state transition denoted by assigning the new state value to a global variable. For easy reference all these functions are declared in a single file. The TDMA state machine has a single entry point for inducing state transitions, the only parameter is the target state. Note that these are states on the MAC layer and the PHY layer provides its own set of states. The state transition function is responsible for which actions to take when the system changes to a certain state, such as changing the radio to RX or TX mode. The code for the function itself describes the operation in fairly simple terms and is listed in listing 7.1.

```
1  void mac_tdma_machine_transition(state_machine_state_t
       target)
2  {
3    ENTER_CRITICAL_REGION();
4    // IDLE is the only state where the radio should be asleep
5    if(target == Idle){
6      mac_trx_sleep();
7    }
8    // Now that the radio is alive, set the proper mode and
         channel
9    switch(target) {
10     case Talk:
11     case Answer:
12       enable_request(TX_STATE,20);
13       break;
14     case Listen:
15     case WaitForAck:
16     case Join:
17       enable_request(RX_STATE,20);
18       break;
19     case Idle:
20       link_scheduler();
21       break;
22   }
23   // And finally we change the current state
24   TDMA_state = target;
25   LEAVE_CRITICAL_REGION();
26 }
```

Listing 7.1: TDMA state transitions

7.1.3 Data Link Layer Join

As specified by the WirelessHART standard, the join sequence (described in section 3.7) will consist of an active search with a timeout where the node listens for any advertisement packet coming from the target PAN. Should no such packet be observed within a configured time limit, the node will transition to passive search mode in order to conserve power.

While in passive search mode, the node will intermittently wake up and listen for a little while before going back to sleep.

When enough advertisement packets have been captured to ensure correct synchronization of the clock, by utilizing the ASN sent in the advertise packets, the node will proceed with the join sequence and utilize the advertised join links to communicate with the network while in join mode.

The initial state and join process of the TDMA state machine can be described by the following piece of commented pseudocode (listing 7.2). The TDMA start function will be called as part of the MAC layer initialization procedure.

```
1  void mac_tdma_machine_start() {
2    // Set initial state to Join
3    mac_tdma_machine_transition(Join);
4    // Start active search
5    mac_tdma_machine_join_active();
6  }
7
8  void mac_tdma_machine_join_active() {
9    // Wait for packets for a fixed period of time (
       ActiveSearchShedTime)
10   // If active search yields no results after a set time,
       revert to passive search
11   mac_tdma_machine_join_passive();
12 }
13
14 void mac_tdma_machine_join_passive() {
15   // Wait for packets
16   // Eventually call the scheduler once the join is a
       success
17   mac_tdma_machine_transition(Idle);
18 }
19
20 void mac_tdma_machine_join_packet_received() {
21   // State is entered when a network packet is received
22   // If the DLPDU is not an ACK, the start time of the
       packet is recorded
23   // Start time of all additional non-ACK packets are
```

```
      compared to the devices slot timing
24   // Synchronized when slot timing statistics converge
25   // Also update network tables
26 }
```

Listing 7.2: Initial state

7.1.4 Link Scheduler

The overall design of the link scheduler has not changed much from the link scheduler implemented by Tegelsrud and Frøysadal. However, the function that serves the time slots after the timer has expired has undergone some modifications in order to make it compatible with the implementation of the TDMA state machine.

As we can see in listing 7.3 the entire function has been wrapped in a critical region in order to prevent spurious interrupts and incoming packets from interfering with the serving of the slot. In addition the state machine transitions involved in serving the slot are implemented here and work by transitioning from `IDLE` mode into either `TALK` or `LISTEN` mode based on the slot type being served, after which the state transitions back into `IDLE` mode. Some details have been left out of the listing in order to make in easier to understand.

```
1  void serve_timeslot_cb()
2  {
3    ENTER_CRITICAL_REGION()
4    if link direction is TX:
5      // TDMA State transition
6      mac_tdma_machine_transition(Talk)
7      if packets are waiting:
8        xmit_frame();
9      else:
10       if time to send advertisement:
11         // First create a basic DLPDU
12         frame = create_dlpdu();
13         // Then make it an advertise DLPDU
14         frame = create_advertise_dlpdu(frame);
```

```
15          // Transmit the frame
16          xmit_frame(frame)
17        else
18          mac_tdma_machine_transition(Idle)
19        endif
20      endif
21    else
22      mac_tdma_machine_transition(Listen)
23      recv_frame();
24    endif
25    LEAVE_CRITICAL_REGION()
```

Listing 7.3: Serving time slots (Pseudocode)

7.1.5 XMIT and RECV Engine

FR01 - A Working XMIT and RECV Engine on the Link Layer (Table 2.1) in section 6.2 states that we need to implement a functioning transmit (XMIT) and receive (RECV) engine on the link layer. This section describes the implementation of the XMIT and RECV engine in order to meet FR01.

The XMIT and RECV engine were mostly implemented when we started our project, however, there were a couple of serious issues that required a lot of debugging effort to locate. During testing we kept running into problems where the node would run out of large buffers. This was due to multiple problems, first we stepped through the code and discovered that buffers were not properly freed after sending packets. Because of this, after a certain number of packets were scheduled for transmission, the loop sending packets would halt due to there not being any more free buffers to build packets in.

After this was fixed we had more problems with running out of buffers and after some investigation we discovered that even though we were sending broadcast packets only and not going into RX mode after sending a packet, interrupts on packet receive were still enabled. The node would proceed to allocate a buffer every time a packet was seen and thus after

some time, exhaust the number of available buffers as we did not process or free any buffers for incoming packets. After disabling the interrupt handler for incoming packets we were able to have two nodes transmitting continuously for over 10000 packets before shutting down the test, which is a significant improvement over the original 16. The fact that we were able to transmit so many packets confirmed that the buffer problem had been solved.

After finding and fixing bugs we moved on to implementing the TDMA state machine and the use of the new RX and TX queues in the XMIT and RECV engine and these have been implemented according to the design specification (Section 6.5.3 and 6.5.4). The coarse outline of these function can be seen in listing 7.4 and 7.5. Note that some of the code has been removed for readability.

In listing 7.4 we see how the sequence number is inserted into the frame and that the frame is built before a timer is started. When the timer expires the frame is transmitted. In listing 7.5 the RECV engine starts its RX timer in order to wait for an incoming packet and aborts if the timer expired.

```
1  void xmit_frame(frame_info_t *transmit_frame, link_t *link )
2  {
3    uint32_t now_time;
4    pal_get_current_time(&now_time);
5    uint64_t asn = GET_CURRENT_ASN(now_time);
6    transmit_frame->seq_num = asn & 0xFF;
7
8    uint8_t *frame = frame_create(transmit_frame);
9
10   // Wait until it's time to send the frame
11   uint32_t TxDelay = TsTxOffset - (now_time - (
         GET_CURRENT_ASN(now_time) * 10000));
12   pal_timer_delay(TxDelay);
13
14   // Send frame
15   data_request(frame);
16 }
```

Listing 7.4: XMIT Engine implementation

```
1  void recv_frame(link_t *link)
2  {
3    uint32_t now_time;
4    pal_get_current_time(&now_time);
5    uint32_t RxDelay = TsRxOffset - (now_time - (
       GET_CURRENT_ASN(now_time) * 10000)) + TsRxWait;
6    // Start the RxWait timer, if no response by
       RxMaxPacketEnd - indicate no response
7    pal_timer_start(TIMER_SERVE_SLOT,
8             RxDelay,
9             TIMEOUT_RELATIVE,
10            (void *)recv_no_response_cb,
11            NULL);
12 }
13
14 void recv_no_response_cb(void *parameter)
15 {
16   // If TDMA_state still is Listen - nothing received
17   if(TDMA_state == Listen) {
18     mac_tdma_machine_transition(Idle);
19   }
20 }
```

Listing 7.5: RECV Engine implementation

7.1.6 Time Synchronization

As stated in FR05 - Time Synchronization (Table 6.1) in section 6.2 we need to be able to synchronize time on all the devices in the WirelessHART network in order to have a functioning network. The following section explains how we implemented the time adjustment functionality.

7.1.6.1 Adjusting the System Time

The system time in AVR2025 consists of the `sys_time` external variable in addition to the `TCNT1` register (Timer1). `sys_time` keeps the most significant bits while the `TCNT1` register contains the least significant part. The PAL library in AVR2025 does not by default contain a function for setting the current system time, thus, this function has been implemented (Listing 7.6).

```
1 void pal_set_current_time(uint32_t time)
2 {
3   TCNT1H = (uint8_t) (((uint16_t) time) >> 8);
4   TCNT1L = (uint8_t) (((uint16_t) time));
5   sys_time = (uint16_t) (time >> SYS_TIME_SHIFT_MASK);
6 }
```

Listing 7.6: Adjusting the system clock

Since `TCNT1` is a 16-bit register and the chip only supports single byte writes we need to write one byte at a time. In addition, the behavior of the write operation is such that if we write the high 8 bits of the register first by using the defined variable `TCNT1H`, and afterwards the `TCNT1L` low 8 bits, the whole 16-bit register will be written automatically when we write to `TCNT1L`.

This approach seems to function correctly when it comes to setting the system time, however, we have yet to investigate if this operation will have any adverse effects on the timers that are already running. If this is possible then we also need to stop and start the timers when the new system time is in effect.

7.1.6.2 Changing Clock Types on the Nodes

As mentioned earlier the nodes have several clock types and one of the more severe problems we had was due to the inaccurate on chip RC Oscillator clock. When we discovered this problem we solicited the help of one of our supervisors, Niels Aakvaag, to assist in working around the problem. After a day spent modifying AVR2025 library code in order to get the

clock source changed, we got nowhere and found out that this had to be more thoroughly investigated. As we looked into how the on-board 32768Hz crystal oscillator (XTAL) could be used, we found that the RTC (Real Time Clock) interface had its configuration registers interfaced by the ATmega3290P micro controller. So in order to get this to work, we would have to modify the kernel running on the ATmega3290P to use the crystal oscillator, and modify the program running on the ATmega1284P to receive clock ticks from another TW (Two Wire) interface instead of its on-board clock since use of the RTC on the ATmega1284P is not currently supported in the AVR2025 library. The magnitude of this task, in addition to the fact that the nodes does not have hardware offloading capabilities for encryption, added to the fact that we would recommend continuations of this project to use other hardware fore wireless nodes. As we mentioned in section 4.1.2 the ATmega128RFA1 one chip solution with the ATmega128 micro controller has both hardware encryption support as well as an on-board crystal oscillator. In this case the clock source can be selected with a dip switch on the STK600 development board as well as with fuses. A transition to this micro controller would most likely solve the problems with both RC oscillator clock inaccuracy and hardware encryption offloading.

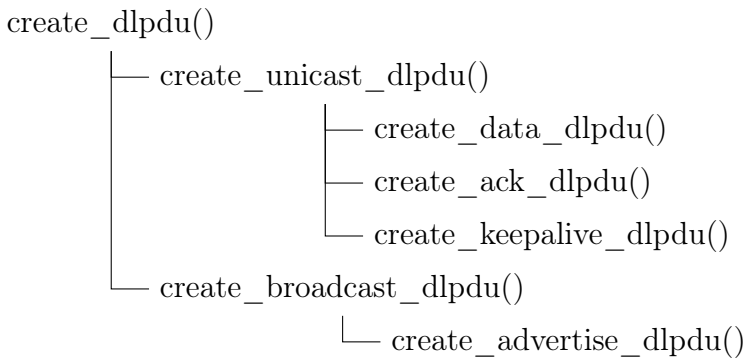
7.1.7 DLPDU Construction

While working on the implementation of the XMIT and RECV engine it quickly became apparent that the previous implementation of creating the various DLPDUs would not suffice in the long term.

In Tegelsrud and Frøysadal's implementation the code to construct and populate data packet structures were tightly integrated with the function that actually transmitted the frame. In order to make this more flexible we have continued to implement the planned abstraction of functions that create DLPDUs in a hierarchical manner.

The primary reason behind this was that there was a lot of code duplication that made debugging and bug fixing very error-prone. The solution to this problem was implementing a hierarchical structure for constructing

DLPDUs as seen below.




```
1 frame_info_t* create_dlpdu(
2   uint8_t npdu_handle,
3   uint8_t *payload,
4   uint8_t npdu_length,
5   dlpdu_priority priority,
6   uint32_t timeout,
7   addressing_mode address_mode,
8   uint64_t destination,
9   uint8_t bcast_flag);
```

Listing 7.7: DLPDU Construction

The root function (Declaration in listing 7.7) is responsible for allocating the buffer and creating a basic DLPDU structure. The pointer to the returned buffer can in turn be passed to the child function in order to populate the packet further with the fields required for that particular DLPDU type. This will minimize the amount of redundant code and in addition to making it very easy to implement changes in the base DLPDU structure across the whole board.

7.2 WirelessHART Gateway

In this section we describe the initial implementation of the WirelessHART Gateway as well as the implementation of the logical subcomponents of the WirelessHART Gateway including the Access Point, Network Manager and

Security Manager. We also describe our attempt to use the Contiki OS with the RavenUSB sticks for implementing an Access Point, issues with it and explain why we in the end had to abandon the idea of using Contiki for our APs 7.1.

In order to implement a functioning Network Manager (FR06 (Table 6.1)) using the RavenUSB stick as an Access Point (AP), with the ATmega1287 micro controller we had to design and implement this from scratch. We decided that implementing it in Java would make for a faster and presumable more substantial implementation than in C. The application was designed with a network stack that was as advanced as the one residing on the nodes at any given time. This meant it would need to have a complete PAL (Physical Abstraction Layer) and a MAC (Medium Access Control) Layer API so that it would be able to receive and send well-formed WirelessHART frames up to the MAC layer. From here, the idea was to implement a rudimentary Network Manager that could construct superframes and administer joins and parts according to the standard.

The standard does not prefer any way of logically separating the modules that is a WirelessHART Gateway. So in our implementation, all the necessary modules are represented as classes within one application.

7.2.1 Access Point

In order to implement a functioning Gateway, the first step is enabling communication between the host computer and the WirelessHART network, which is also what FR04 - Interface Between Computer and Sensor Nodes (Table 6.1) in section 6.2 describes. This creates the need for an Access Point and the first step in this process was to actually be able to talk to the network via a suitable device. With the equipment we had the RavenUSB stick was an obvious choice. The challenge was that we had no documentation of the software residing on the RavenUSB stick, and when contacting Atmel the only reply we got regarding this was that the software was made by a third party supplier and that they had no documentation to provide us with.

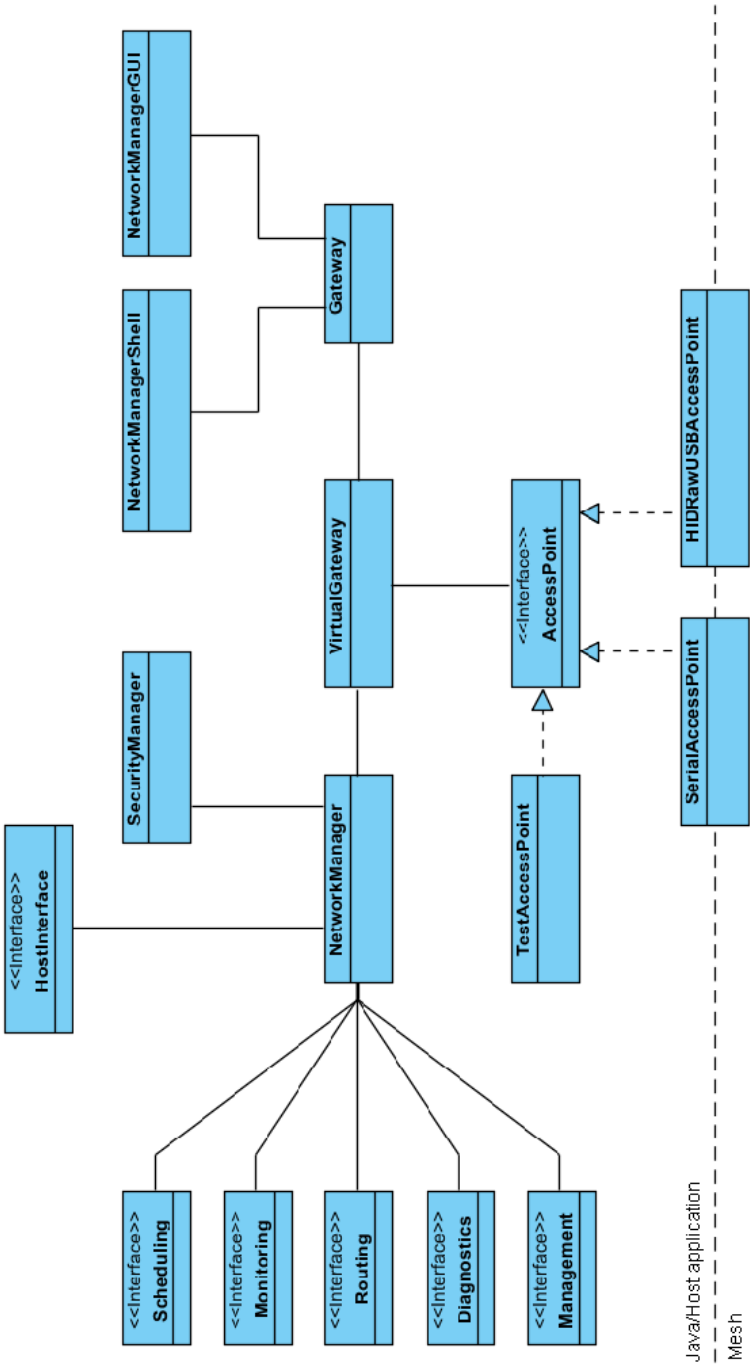


Figure 7.1: Gateway Class Diagram - Showing the classes and interfaces that makes up the Gateway. It also shows the Access Point interface instantiated with two different sets of physical layers.

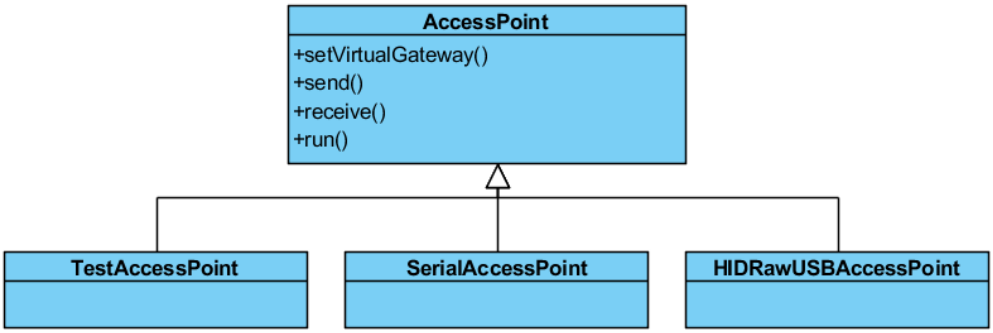


Figure 7.2: Class Diagram of Access Point - This figure shows the Access Point as an interface that can be extended to match Access Points that work on different technologies.

In order to interface this device, we had to investigate several open source alternatives. The figure 7.2 shows how an Access Point is built down to the physical layer, where the interface towards the RavenUSB stick constitutes as the Access Point PAL (Physical Abstraction Layer).

7.2.1.1 The Contiki OS

The Contiki platform is an open source, modular operating system for low power consuming embedded systems. Its primary goal is to offer a fully functional IPv4 and IPV6 stack on an embedded system, where Atmels Raven Avr devices are several of the supported units. The development has been lead by developers from the Swedish Institute of Computer Science, but are supported by contributors from Cisco, Redwire LLC, SAP, SICS, and others.

Contiki Development Model

The entire Contiki OS is written in the C programming language and offers a small event-driven proto-threaded (low overhead threading technique) kernel with the option of adding modules that can be dynamically loaded during runtime. These modules are added based on application requirements and ROM / RAM availability.

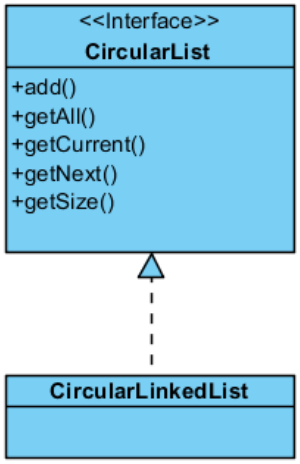


Figure 7.3: Class Diagram of our Circular List interface - Whenever implementing a queue-list this is used in order to insure deterministic list operation.

Adapting Contiki to Support WirelessHART

The Contiki OS is a large project and as seen in figure 7.4 the most recent commit (September 2011) consists of 573,870 lines of code and 272,420 lines with comments. Needless to say this is a huge amount of code and it therefore took a considerable amount of time to familiarize ourselves with the Contiki OS, the reason behind being that we originally wanted to use it for implementing the Access Points on the RavenUSB stick.

After successfully using the RavenUSB sticks with Contiki as sniffers we started looking at using them for our Access Point as well. Among other standards Contiki supports the IEEE802.15.4 standard. We wanted to combine Contiki and the RavenUSB sticks in order to create Access Points for our Network Manager, however we could not do this with the OS as it was, since it was programmed for the IEEE802.15.4 standard IPV6 over Low power Wireless Personal Area Networks (6LoWPAN)[28]. Due to the Contiki OS using this 802.15.4 standard the whole setup and capturing of packets was adapted to 6LoWPAN, meaning the packets were modified in order to meet 6LoWPAN requirements.

In order to send data using the RavenUSB sticks we made a small

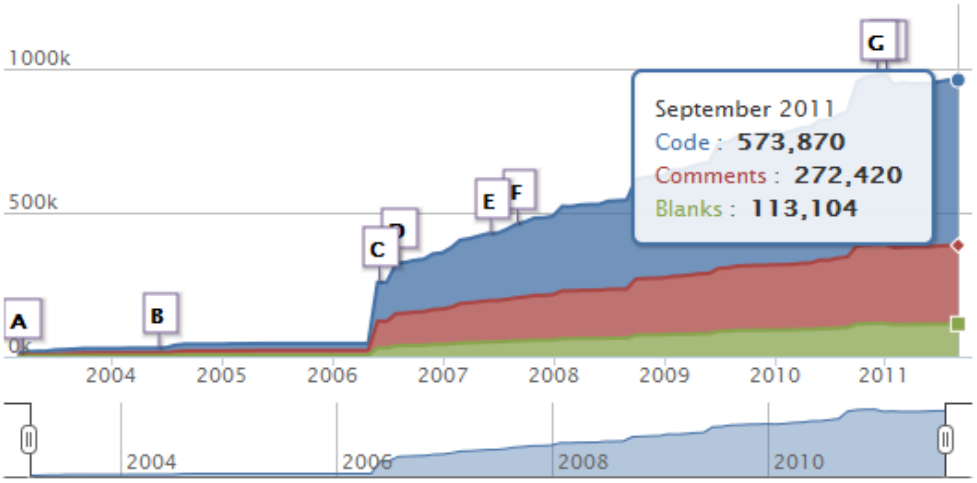


Figure 7.4: Contiki OS code analysis [11] - Provides an overview over the amount of code lines in the Contiki code base per September 2011.

Python test program which bound to a raw socket to the USB stick, filled the buffer with a valid WirelessHART header, payload and a message integrity code (MIC) and then sent this buffer to the socket. However, the packets we sent would be modified by Contiki making it impossible to set the DLPDU specifier, using short address mode and it would change the payload by overwriting the first few bytes. This made us unable to control how the format of the data we were sending and we therefore needed to adapt the Contiki OS to meet our demands.

Address Mode Problem

The original Contiki code we downloaded used the long address mode (8 bytes) by default, but it did also have support for the short address mode (2 bytes). The header file `frame802154.h` defines `FRAME802154_SHORTADDRMODE` and `FRAME802154_LONGADDRMODE` which indicates if long or short addresses are to be used in the data sent. In `sicslowmac.c`'s `send_packet()` function the different attributes of the packet to be sent was set and this is where Contiki by default set all the addresses to be in the long address mode and also read 8 bytes from the input buffer in stead of 2.

In order to change this we set the source and destination address modes to `FRAME802154_SHORTADDRMODE` in stead of `FRAME802154_LONGADDRMODE` as default. In addition to this we read 2 bytes instead of 8 from the buffer and this solved the problem. Contiki now uses short address mode and sends the addresses as expected.

Although this solved our problem a better fix would be to read the address specifier in order to see which types of address mode we want to use, choosing the mode according to this, and then read the correct amount of bytes for the source and destination.

Payload Issues

Originally, when sending a test packet with Contiki, the first few bytes of the payload would be overwritten by "junk". The reason for this was Contiki expecting a 6LoWPAN packet, where the header is larger than a WirelessHART header [38, 2.3] and therefore overwriting the payload with bytes it intended to add to the 6LoWPAN header.

We tried to fix the problem in several different ways, starting by changing the `PACKETBUF_HDR_SIZE` to the header size of our WirelessHART packet, which with short source and destination addresses was 10 bytes in stead of the original 48 bytes. This was done because when adding the payload to the frame Contiki was building in `send_packet`, the payload part of the frame was set to `&packetbuf[bufptr + PACKETBUF_HDR_SIZE]`. This however made no difference to the payload displayed in Wireshark, so we had to try something else.

The next fix we tried was to go to the packet queue and try and get a hold of the original buffer with raw data from RNDIS (Remote Network Driver Interface Specification) and then creating a pointer to the buffer so we could access it in `sicslowmac.c`'s `send_packet()`. When trying to point to this data instead for the payload part of the frame we only got junk and nothing was left of the original payload. After spending a lot more time than anticipated and still not finding a way to fix this problem we decided to just ignore the first few bytes of the payload and make the nodes in the

network pretend the first few bytes of the payload did not exist and the payload started at for instance byte 20 in stead of byte 11.

Ideally we would have liked to get the payload problem fixed properly, where the payload would begin at the anticipated position and formatted as expected. However we had already spent a considerable amount of time on Contiki and were eager to get on with other parts of the project and therefore decided to try and improve this at a later occasion if we had the time.

DLPDU Specifier

6LoWPAN's header does not include a DLPDU specifier and Contiki therefore did not expect a DLPDU specifier. This resulted in the DLPDU specifier included in the packet sent from the python test program being ignored by Contiki and replaced with another value, thus making it impossible to set the DLPDU specifier.

The first thing we did to fix this was to check the `send_packet()` function in `sicslowmac.c` to see if it was possible to just set the DLPDU specifier byte statically in the buffer. Setting the byte right after the source address to a legal DLPDU value worked and showed up in Wireshark correctly. The new goal was to find where the DLPDU specifier value was inserted in the header by the test program and insert this at the same place in order to choose different DLPDU specifier values. It seemed to be as simple as getting the correct byte out of the buffer sent to the USB stick via RNDIS, however this was not the case. The buffer was configured several times in different layers in Contiki, formatting it so get-functions could be used in order to get the attributes and addresses one was looking for from the buffer and these were of course made for 6LoWPAN and not for WirelessHART packets. This made it impossible to just go through every byte to find the right value and we realized the task would be a lot more time consuming than first anticipated.

After the initial changes we were now able to set a static DLPDU specifier, however that was not enough since it would be impossible to

actually use Contiki if we could not configure the specifier. It would have limited us to only send one type of packets, for instance only data packets or only join packets. At this point we decided we either had to spend a lot more time on Contiki than we had hoped or we needed to find an alternative to Contiki.

Problems With Interfacing the Contiki OS

In order to communicate with the RavenUSB Stick we have to open an interface to the device node the Contiki identified itself as. And since the Contiki OS uses NDIS (Network Driver Interface Specification) drivers via NDISWrapper [10] the to identify itself to the host OS we could use Berkeley sockets to connect to the device node, which was usb0 in our case.

To open a socket with the earlier mentioned properties we use the following line `sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))`. The definition of the function is `sock = socket(DOMAIN,SOCKETTYPE,ETHERTYPE)`, so we are asking for a socket where as little as possible is done by the kernel. More specifically no intervention from any kernel level flow control or routing protocol.

The PF(Protocol Family) family domain in Linux defines a way of communicating directly with the network adapter and the `SOCK_RAW` defines that no IP or datagram header info should be appended by the kernel. Further we have to choose what ethertype we want to communicate with, and this is set to a macro that would allow all traffic to pass, as opposed to the default setting that only allowed Ethernet packet headers. In order to prove the concept we designed a short (approximately 100 lines) of Python code to send a simple and static WirelessHART frame over the RavenUSB stick.

The problem at hand arose when it became clear that current Java Development Kit (JDK) 1.7 does not support creation of raw sockets. Since this obviously worked in Python, the usage of the Java Python combination Jython was interesting due to the ease of implementing the desired sockets in python. Unfortunately Jython currently does not support `PF_PACKET` or

`AF_PACKET` socket types. This also contributed to the decision to abandon the Contiki OS and use the 15dot4 tools. Since this OS identified itself as a HIDRAW device, and made working around the Linux network stack unnecessary.

Abandoning Contiki

As seen in the past few sections Contiki was getting a lot more than we could handle in the time provided and we therefore decided to start looking for alternatives. The Contiki OS consists of more than 500 000 lines of code and several hundred files and we realized it would be a project of its own to adapt the OS to WirelessHART. The problems we encountered with sending packets were too great to ignore and even though we could adapt to the payload problem we could not ignore the inability to change the DLPDU specifier as it is a crucial part of a WirelessHART packet. However, we also knew we could not start using the Network Manager without any Access Points, so it was crucial that we found a way to properly communicate with the RavenUSB sticks in order to use its radio. After researching a while for new software for the RavenUSB sticks we finally found a promising alternative to Contiki; the 15dot4-tools Project.

7.2.1.2 The 15dot4-tools Project

Since Contiki seemed to become too large to handle in our limited amount of time we decided to look for other options. In this process we found the website of Colin O'Flynn[1], one of the Contiki contributors, who has also made his own open source 15dot4-tools Project. This project basically contains different 802.15.4 tools such as an 802.15.4 Sniffer, 802.15.4 Raw Packet RX/TX Tools and 802.15.4 Wireshark Packet Analysis (detective). Most of these he only wrote for the Dresden (now owned by Atmel) SAM7S USB stick[14]; however the sniffer tool could be programmed on the RavenUSB stick.

In addition to the actual code for the sniffer the project also provides a configuration program which can be run in order to change the chan-

nel of the radio and change the mode of the RavenUSB stick, e.g. from sniffer to bridge. It also provides a debug mode which opens a terminal allowing the user to print out registers by giving commands to the sniffer program through the serial port. Except for this, communication between the RavenUSB stick and the host device is done through RNDIS.

With the sniffer software running on the RavenUSB stick it will identify itself as an Ethernet interface on the host computer, for instance eth1. In order to set the desired channel the configuration program needs to be run with the command `./ravenusb -d DEVICENAME -c CHANNEL`, so in Linux it would be `./ravenusb -d /dev/hidraw1 -c 20` in order to set the channel to 20. After plugging the Raven USB Stick in and setting it to the right channel it is ready to sniff packets and in order to display these we can use Wireshark with eth1 as the capture interface.

Adapting the 15dot4-tools Sniffer to an Access Point

Since we intended to use the RavenUSB sticks as Access Points for our Network Manager we had to change the sniffer code to fit our requirements. The RavenUSB stick is only used for its radio, so the Access Point part of the Network Manager is only used to communicate with the RavenUSB stick when it wishes to send something and listen for any incoming packets.

Since we decided to write the Network Manager and all its components in Java and communication with the RavenUSB stick is done using a HIDraw (Human Interface Devices) device, we needed a HID communication API in Java. We also wanted to run this on a Linux machine, making the HID communication tools Atmel provides useless, as these only work for Windows. This will be described in detail in the following section.

7.2.1.3 Enabling the Raven USB Stick in Linux

In order to communicate with the RavenUSB stick from a Linux machine, we had to have a way of communicating with the RavenUSB driver. This section focuses on how we enabled this communication and how it works.

The HIDraw Device Driver

The HIDraw driver in Linux was intended for use with USB Human Interface Devices (HID) that do not need to be parsed or altered in the HID module of the Linux kernel, thus making it applicable to user space applications that had a precise definition of how to communicate with the hardware at hand. This is done either by utilizing user space drivers or through direct communication between the HID unit and the application. This driver type differs from the hiddev driver type that implies in-kernel parsing and queuing of data to and from the the device.

The main reason for us utilizing this driver type in this project is that it extends a relatively intuitive API, and that the 15dot4 tools already had an HIDraw interface to the host that could be modified and extended to support sending and receiving of packets.

The HIDraw API

The HIDraw API[35] implements a vast amount of functions but we are only using a subset of these in this assignment.

- `open()` This function opens and prepares a device descriptor for the HID device.
- `read()` This function reads a queued report from the device. This can happened either in a blocking or non blocking fashion based on flags sent in the `open()` function.
- `write()` This function will write a report to the device.
- `ioctl()` The ioctl function (abbreviation of input/output control) uses system calls to send device-specific operations to set or retrieves a set of parameters identifying properties on the device. A subset of these operations, which are in use in our application follows:
  - `HIDIOCGRDESCSIZE` This will supply the length of the device descriptor in byes (i.e. `/dev/hidraw0`)

- `HIDIOCGRAWINFO` This will report a struct containing the Vendor ID (VID), the Product ID (PID) and the bus type (i.e. USB, Bluetooth or virtual)
- `HIDIOCGRAWNAME` This will return an UTF-8 string containing the vendor-defined name of the device
- `HIDIOCGRAWPHYS` This will retrieve the physical address of the device. In USB devices this will contain the physical path to the device. This is a concatenation of the id's of the USB controller, USB hub, port, endpoint etc. In Bluetooth devices, this can be the MAC address.

The JavaHIDAPI Project

Given that we wanted to implement the Network Manager in Java, we Needed a way to communicate with the C/C++ based HIDAPI library through Java. The obvious choice was to wrap the functions with Java Native Interface (JNI) (Java Native Interface) which instantiates C/C++ methods/objects as Java classes with subordinate methods which corresponds to the original HID library. After some investigation we found an open source project called Java HID API[6]. This project constitutes of a jar file that contains a library of methods to list, open, instantiate, send and receive from HID device nodes. We used this in a non-blocking fashion on the physical layer of our Access Point. This proved to work fine as long as a HID driver based OS is running on the Access Point's radio controller.

Suitability

It is an inherent issue that the USB based radio (On the RavenUSB stick) is not a human interface device, and that the driver type used is not ideal for this purpose. It works because WirelessHART packets are small, and that the bandwidth needed is very limited. But for future use other device drivers should be considered.

7.2.1.4 Sending and Receiving Packets on the Access Points

This section describes how the sending and receiving functionality has been implemented on the APs.

Receiving Packets

When the RavenUSB, with 15dot4 tools, is not currently doing anything it is constantly listening for packets (in RX mode). If one is received it is sent via HID regardless of whether the other end is listening or not (polling the device).

The Access Point (AP) will only be polling the HID device (RavenUSB) when it is in an RX slot. The Access Points' Application Layer constantly checks its list of links to determine which link is the next to be served and whether it is an RX or TX link. In the event of an RX link the receiving process starts. When the link slot in the superframe has been reached the Access Points' Application Layer calls the Session Layer receive function. This in turn calls the Transport Layer, which calls the Network Layer and so on until the Physical Layer has been reached. This is where the actual polling of the HID device starts.

The JavaHIDAPI supports two types of polling of the HID device; blocking or non-blocking. In order to combine the polling of the HID device with RX-slots, we knew we could not block while reading. The reason for this is that we should only poll the radio for as long as the radio should be listening in a slot. When blocking the Access Point would have to wait until it receives something from the HID device before resuming to its other activities. This would lead to problems if a packet is not received in the RX slot, causing the Access Point continue blocking through other slots which is not acceptable. We therefore decided to set the HID read in non-blocking mode and read in a while-loop for the amount of time dictated by the WirelessHART standard. If a packet has been received during an RX slot, the return value of the HID-polling will not be null and the Access Point will therefore stop the polling and return the received frame. The received frame is then sent back up through the layers and

when it reaches the Application Layer the packet will be put in the event queue of the Virtual Gateway for further processing.

Sending Packets

The Gateway prepares a packet with all the information needed and places it in the shared TX queue. This is a First In First Out (FIFO) queue, where the oldest packets will be sent first. Now it is the Access Points' turn to actually send these packets. As explained in the previous section the Access Point will constantly check which link is next. If this one is a TX link the Access Point will go through the TX queue to find the first packet where the destination of the packet matches the neighbor connected through that link. When a packet has been found in the TX queue it is sent through the layers down to the Physical layer where it is sent to the HID device (RavenUSB).

15dot4 tools receives this data through HID and relays this data to its send function which sends the data over the RavenUSB stick radio. If sending the packet was successful the Access Point can delete the packet from the TX queue and continue going through its links. If not the packet stays in the queue and the Access Point will try again next time it reaches that TX slot.

7.2.1.5 Packet Creation on the Access Points

As the 15dot4 based interface between the Access Point PAL (Physical Abstraction Layer) and the RavenUSB only receives a byte-array so the packets has to be created in the Access Point network stack. When sent to the HIDraw device node the RavenUSB sends the byte array unaltered on to its radio.

Data Link Layer on the Access Point

The DLPDU (Data Link Protocol Data Unit) is implemented as an interface that is implemented in the classes DataDLPDU, DisconnectDLPDU, KeepAliveDLPDU, AckDLPDU and AdvertiseDLPDU. These classes each

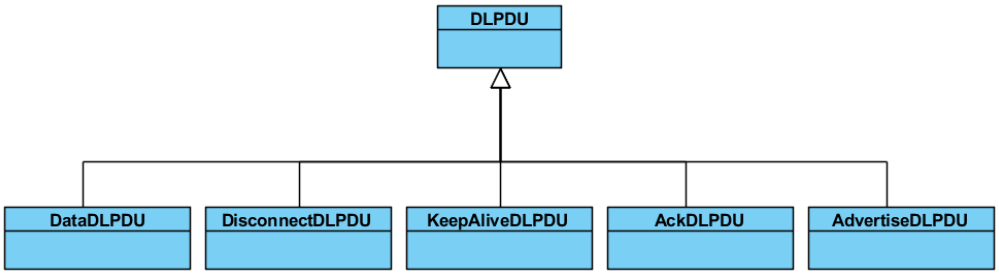


Figure 7.5: Class Diagram - Showing the various available DLPDU types on the link layer.

has a set of header fields, implemented as Byte array variables with the correct length, that corresponds to their name 7.5 and can be used by the above layer. Only the DataDLPDU contains a payload field that can contain above layers.

Network Abstraction Layer and Above on Access Point

The above layers has also been implemented as classes, but not as interfaced classes. Each contains a set of header fields and a payload field corresponding to the standard. Here the header fields have also been implemented as byte arrays with the length that corresponds to the standard.

7.2.2 Network Manager

This section will describe the Network Manager we have implemented and which features have been implemented and which ones still need to be implemented. We will start with how all the components of the Network Manager and Gateway interact with each other and are set up and continue with the implementation of the command interface, scheduling, link selection and the Graphical User Interface (GUI).

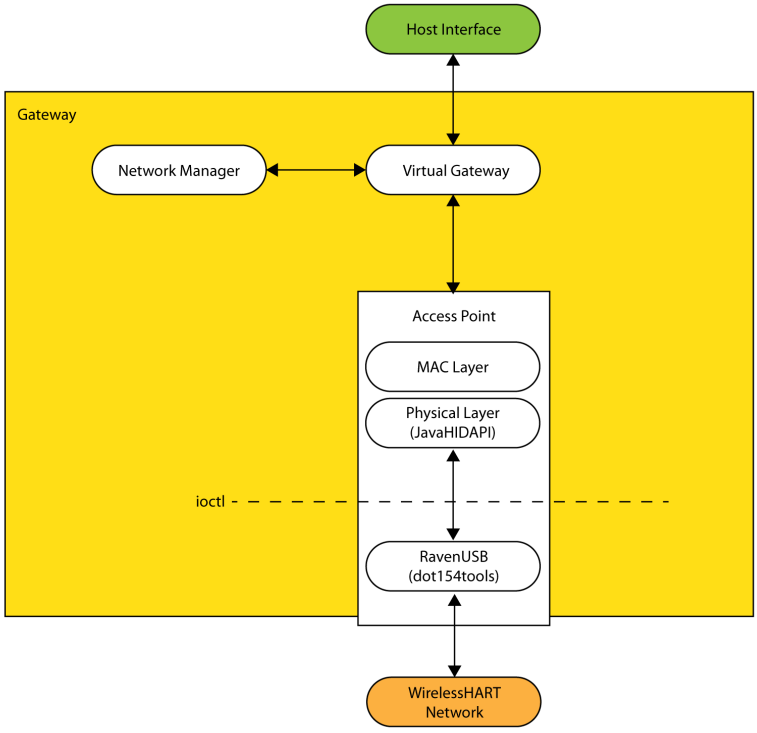


Figure 7.6: Communication between Gateway and WirelessHART Network

7.2.2.1 Communication Between Gateway, Network Manager and Access Points

When starting the program `main()` is located in the Gateway class. The Gateway now start initializing several other classes, setting up the base of the Network Manager, Security Manager and Virtual Gateway (VG), as seen in Listing 7.8. It will also add all Access Points to an array list which it sends to the Virtual Gateway when it initializes it and as of now we only use one Access Point, the `HIDRawUSBAccessPoint`, which communicates with the RavenUSB stick running 15dot4 tools.

After these classes have been initialized the Gateway starts the Virtual Gateway, which already in its constructor started all the APs it received from the Gateway and initialized the RX and TX queues. The Virtual Gateways' `run()` function then creates test packets (three every three seconds) which it places in the TX queue so there is always something to be sent. Meanwhile the Access Point is constantly checking its list of links for the next TX or RX slot and scheduling these according to the current superframe. The sending and receiving of packets have been described in more detail in Section 7.2.1.4.

Figure 7.6 gives an overview of the communication between the different components of the Gateway.

```
1  public static void main(String[] args) throws
       InterruptedException {
2
3      // Initializing GUI
4      window = new NetworkManagerApplicationGUI();
5
6      // Configuring Logger Globals
7      Logger.setHandler(new LogHandler(window));
8
9      // Initializing Command Shell
10     nmShell = new NetManShell();
11
12     // Initiating Gateway Logger
13     logger = Logger.getLogger("gateway");
14
```

```
15      // Instantiate the network manager
16      NetworkManager netMan = new NetworkManager();
17
18      // Instantiate the Security Manager
19      SecurityManager secMan = new SecurityManager();
20
21      // Instantiate the access point(s)
22      List<AccessPoint> ap = new ArrayList<AccessPoint>();
23      a.add(new HIDRawUSBAccessPoint());
24
25
26      // Instantiate the virtual gateway
27      VirtualGateway virtGW = new VirtualGateway(netMan, ap,
            secMan);
28      virtGW.start();
29
30      .....
31  }
```

Listing 7.8: Initializing part of the main method located in the Gateway class.

7.2.2.2 HART Commands Interface

In order to provide a clean interface to HART commands and make it easy to extend by adding further commands at a later stage, we have implemented a command to function map that helps with mapping command codes to functions. The Command object displayed in listing 7.9 represents the map. As we can see in the listing, there are two example commands implemented, the functions for reading and writing the network tag along with their appropriate command number as per the WirelessHART standard. This may later be extended to support multiple command classes through the usage of "Class.Function" mapping as opposed to pure "Function" mapping, meaning the class that provides the function may be a part of the map.

```
1  public enum Command {
2
```

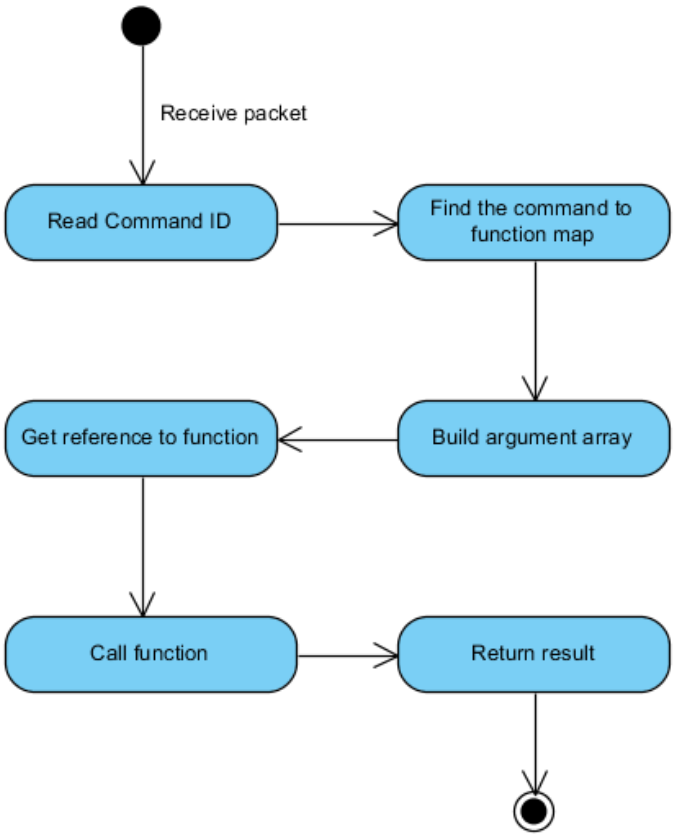


Figure 7.7: HART Commands Interface

```
3   WRITE_NETWORK_TAG(775, "writeNetworkTag"),
4   READ_NETWORK_TAG (776, "readNetworkTag");
5
6   private final int id;
7   private final String fn;
8
9   Command(int id, String fn){
10    this.id = id;
11    this.fn = fn;
12  }
13
14  public int id() {
15    return id;
16  }
17
18  public String fn() {
19    return fn;
20  }
21
22  public static Command getById(int id) {
23    for(Command c : Command.values()){
24      if(id == c.id()){
25        return c;
26      }
27    }
28    return null;
29  }
30}
```

Listing 7.9: Command Enumeration Object

In listing 7.10 we can see the opposite end of the map, where the actual functions are declared. In order to provide an overview of how it works, as previously explained, only two functions are implemented. At a later stage this class will contain all the functions necessary.

```
1 public class VirtualGatewayCommands
2 {
3   public void writeNetworkTag(String tag)
4   {
```

```
5     System.out.println("writeNetworkTag" + "(" + tag + ")");
6   }
7
8   public String readNetworkTag()
9   {
10    System.out.print("readNetworkTag");
11    return "CAFE";
12  }
13 }
```

Listing 7.10: Command Function Map

In listing 7.11 we can see how it all ties together. The first thing that happens is that we get an instance of the command class, before we look up which method to call in this class based on the command number in the Command object given as an input argument. This function is called and the result is returned to the caller.

```
1  public Object execute(Command cmd, Object... args)
2  {
3    try {
4      // First get a class reference and instantiate it
5      Class<?> c = Class.forName("gateway.
           VirtualGatewayCommands");
6      Object t = c.newInstance();
7
8      // Build an argument type array if we have any arguments
9      Class<?>[] cls = null;
10     if (args != null) {
11       cls = new Class<?>[args.length];
12       for (int i = 0; i < args.length; i++) {
13         cls[i] = args[i].getClass();
14       }
15     }
16
17     // Now grab the command to execute and find the function
           name
18     Method m = c.getMethod(cmd.fn(), (Class<?>[]) cls);
19
20     // Execute the function
```

```
21      Object o = m.invoke(t, args);
22      return o;
23
24    } catch (Exception e) {
25      e.printStackTrace();
26    }
27    return null;
28  }
```

Listing 7.11: Command Execution

Using reflection to dynamically decide with class and function to call during runtime makes this very flexible. In order to extend this implementation to support multiple command classes all that needs implementing is an algorithm for selecting and instantiating the proper class based on the information available in the Command object. An example of separation into multiple command classes is to separate the commands of the Network Manager, Virtual Gateway and Local Management into their own classes.

7.2.2.3 Scheduling and Link Selection

During implementation of the Network Manager it became apparent that the original constructs used for selecting the next link to be scheduled for processing implemented on the nodes was quite inefficient. The original data structure was a simple linked list (Figure 7.8) which was then iterated in order to find the next link. This meant that every time we need to look for the next link to schedule, we had to iterate over the list of links and find the next link with the slot number higher than the current ASN for every superframe. This happened regularly at the end of a slot where time requirements are quite critical and should be performed as effective as possible.

In order to do this more efficiently we implemented the superframe links structure as a circular linked list (Figure 7.9) with a current pointer. This means that every time we need to find the next link to schedule in a particular superframe, all we have to do is iterate the current pointer one step forward. When the current pointer reaches the end of the superframe

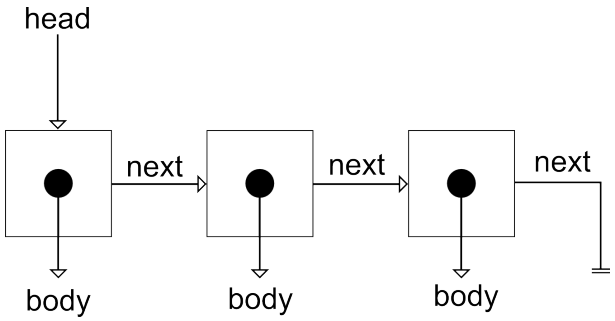


Figure 7.8: Linked list

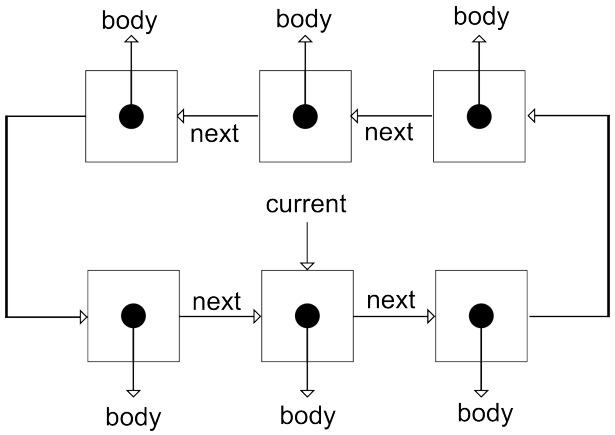


Figure 7.9: Circular Linked list

it will automatically wrap around to the first link. This data structure has been implemented in the Network Manager with an API 7.3 and works quite well and it should also be implemented on the nodes themselves. This is a data structure that will only change when a new and updated superframe is received by the Network Manager so the complexity of inserting and removing nodes from the links is negligible compared to the complexity needed to find the next link to process.

Once the next link to be served has been established based on all the next links in each superframe, and the link scheduling algorithm has selected a link, the Absolute Slot Number (ASN) for this link will be calculated. The node will then wait until it is time to serve the link based on the relative difference to the current ASN.

This can be extended to support multiple superframes as defined by the WirelessHART standard by creating an array of circular linked lists where each of the lists manage one superframe. In order to find the next link to process we then have to iterate over this array which in itself is an $O(n)$ where n refers to the number of active superframes, the link selection itself is an $O(1)$ operation. The algorithm used is described in pseudocode in listing 7.12.

```
1  possible links = array()
2
3  for each active superframe:
4    iterate the circular linked list and find the next link
5    assign this link to the possible_links array
6
7  for each possible_link:
8    use the link scheduler algorithm to compare with the other
         links
9
10 schedule link
```

Listing 7.12: Link selection algorithm

7.2.2.4 Graphical User Interface (GUI)

After implementing large portions of the Gateways' logical components, it became obvious that the amount of information given out in one single console was too much for a user to read efficiently. In addition to this debugging became troublesome for the same apparent reason. We then started to develop a rudimentary GUI that separated the required info into different tabs so that the user was able to focus on the information was useful - without excluding information that potentially could have significance.

The GUI is written in the Swing framework without modifications to the Look and Feel and is tested and developed in JRE (Java Runtime Engine) 7u5.

Control

We implemented what we call the control view (Figure 7.10). This gives the user a real-time view of the network hierarchy. Each device has its own pane which is intended to display device specific variables and constants. The data structure for this view is the internal data structure of the Gateway which has been fitted with an API. This allows us to get a list of all adjacent devices and their children. This will typically be the Network Manager, the Access Points and all the field devices that are neighboring the given Access Point.

The tree structured list is generated each time the tree is expanded, and destroyed when the tree is collapsed. In this way we can assure that any differences instigated during runtime will be visible.

Graph

The graph tab of the GUI is for displaying a graph of the entire WirelessHART network, with all its members from the Network Manager to the Field Devices. The point of the graph is to give an overview of the network and all its components to the user. At the bottom of the panel is a refresh button and each time this button is pressed an updated graph is generated and displayed.

For the implementation of this feature we looked at several possibilities of graph generation in Java. After looking through the possibilities we found that using the textual graph description language DOT to be the simplest way to go. DOT allows the user to describe a graph using simple commands and can later convert this description into an image. However, since we wished to generate the graph from Java, we had to find a way of creating this code dynamically in Java to then execute it with dot and import the image into our GUI. After a little research we found Graphviz[3], which is an open source graph visualization tool. Graphviz has made a Java API that generates DOT code and lets define a graph, choose what type of image you want to create and then executes your DOT code in dot (Linux tool for compiling DOT code) in order to generate the image.

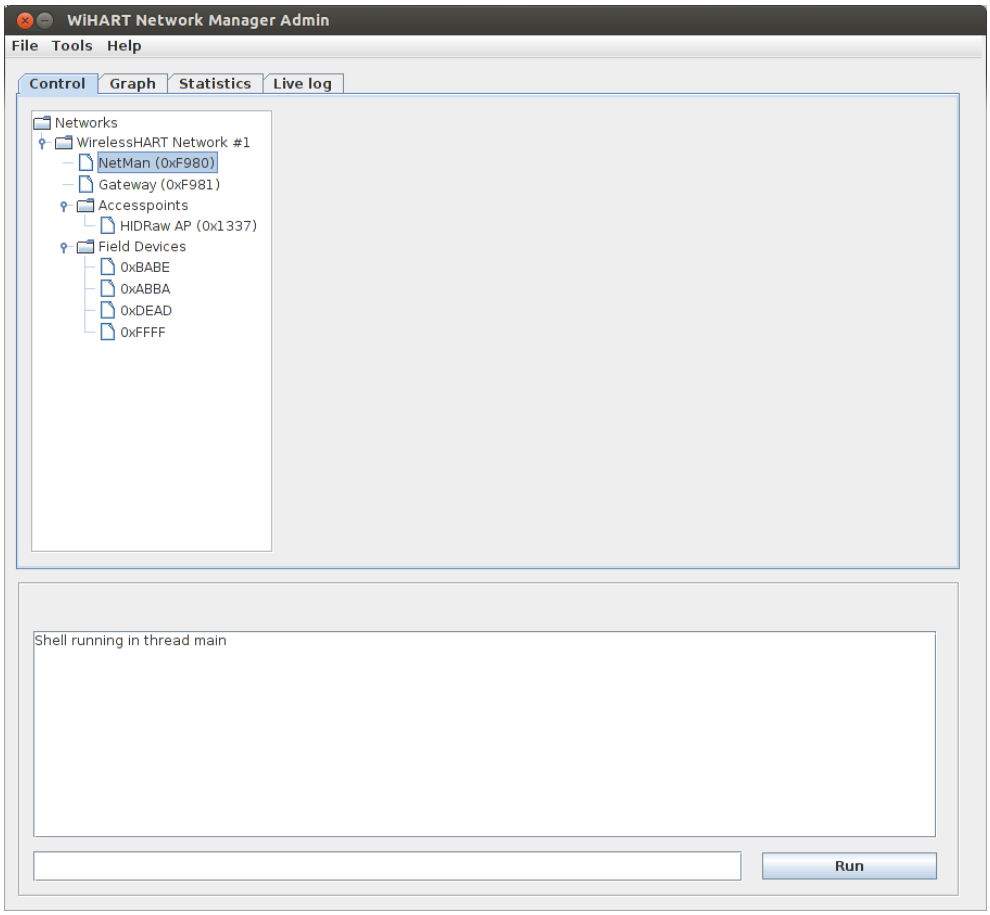


Figure 7.10: The control tab of the GUI

```
1 strict digraph GUIGraph {
2   F980->F981 [dir="both"]
3   F981->1337 [dir="both"]
4   1337->BABE [dir="both"]
5   1337->ABBA [dir="both"]
6   1337->DEAD [dir="both"]
7   1337->FFFF [dir="both"]
8 }
```

Listing 7.13: DOT Code to generate a graph

When the user clicks the refresh button in the graph panel the action listener for the button is activated. The action listener calls the graph generating method `makeGraph()` in the `WHARTGraph` class. `makeGraph()` starts the DOT graph and then calls `dotFileBuilder()`. `dotFileBuilder()`'s job is to plot in the actual nodes and links into the graph. It iterates through all the Access Points and adds these as neighbors to the Gateway. For each Access Point it will then add all its neighbors as nodes in the graph with links between them. After this the `dotFileBuilder()` returns and `makeGraph()` ends the dot graph string and sends it to Graphviz's `writeGraphToFile()` which builds the graph and then writes it to a PNG file which is displayed in the GUI. The DOT code needed to create the graph shown in figure 7.11 is shown in listing 7.13.

Statistics

In order to simplify the collection of statistics from each Gateway component we implemented generic way for all components to store statistics. Each component instantiates a class called `StatHandler` that contains a map of objects of the class `StatEvent`. This class holds two fields, one serving as a description and the other as the adjacent value. A typical event can be `sent_bytes` and this is updated each time something is sent from an Access Point.

When the statistics tab is generated, it retrieves all the `StatEvent` objects from all the Gateway components and both displays and refreshes

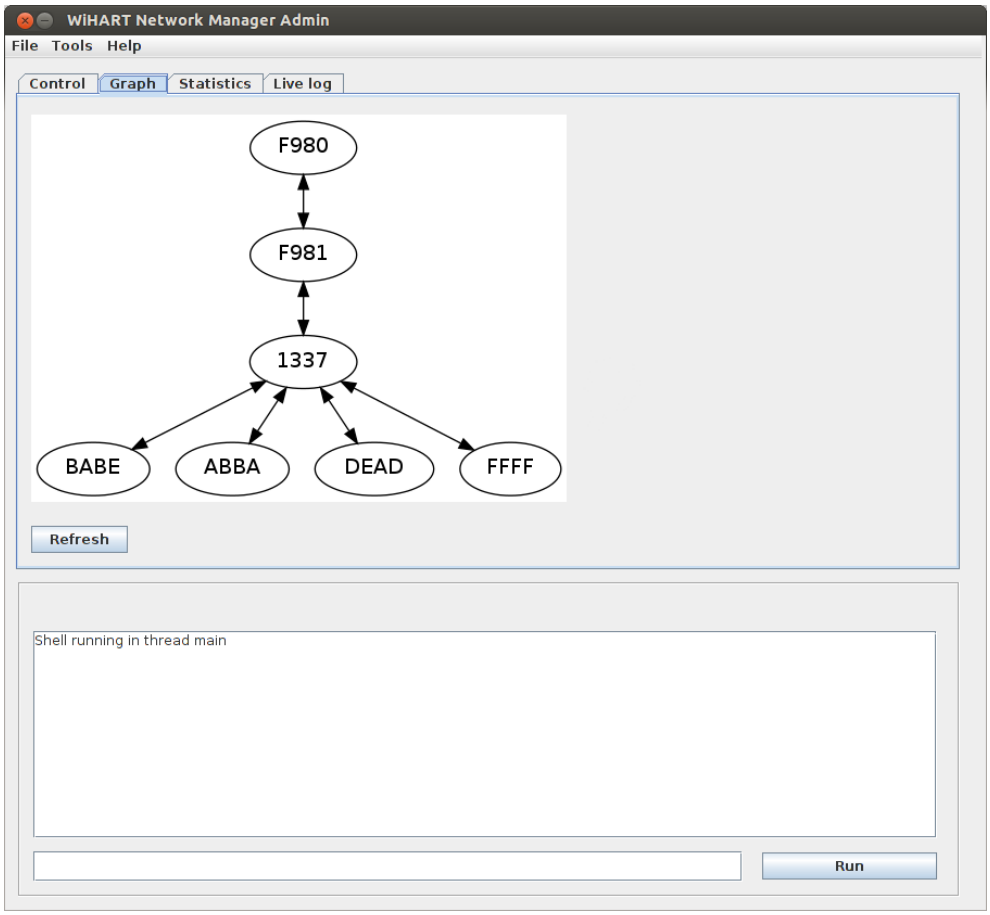


Figure 7.11: The graph tab of the GUI

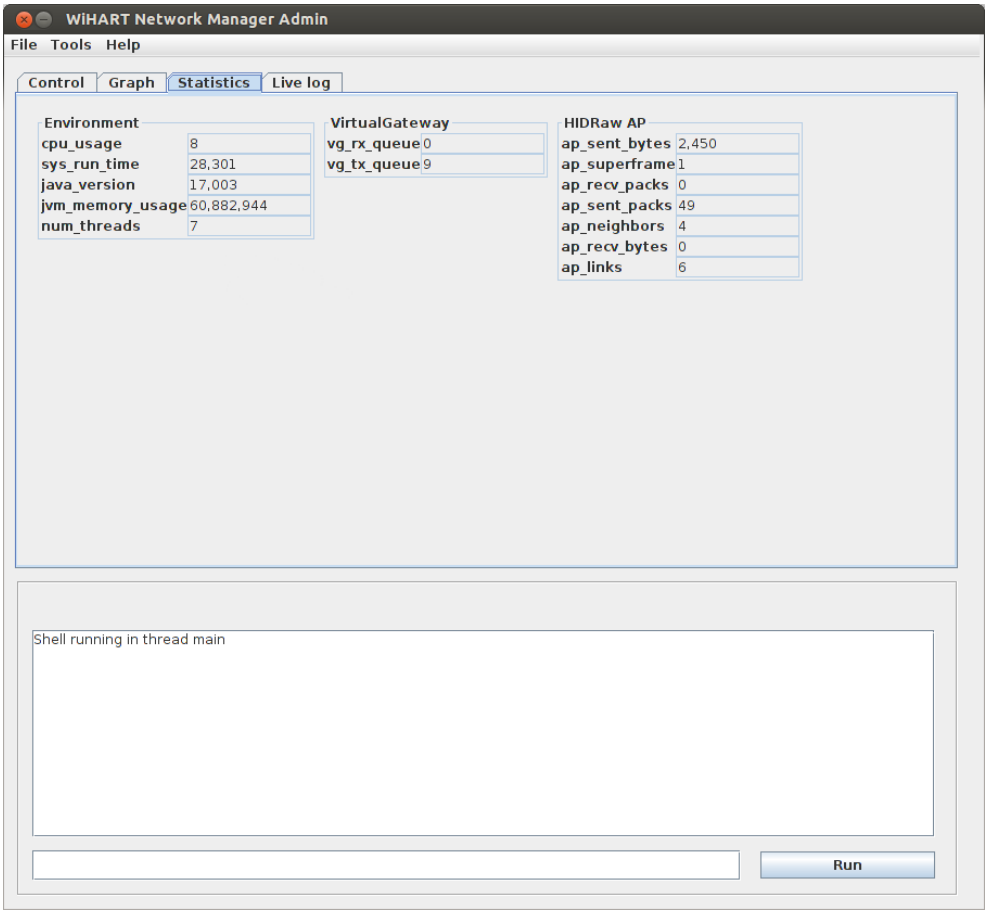


Figure 7.12: The statistics tab of the GUI

them according to a given time interval (Figure 7.12).

Live Incident Log

In order to keep track of the life cycle of the Gateway component, we implemented a logging function that was used in all the Gateway classes 7.13. When each Access Point, the Network Manager, the Security Manager or any other Gateway component started, failed or ended gracefully it would log this as an event with a string representing what the event was and a appropriate log level. These are `INFO`, `WARN` and `FAIL`. In addition the logger will track what method it was called from and finally concatenate these into a log-string that is viewed by the user.

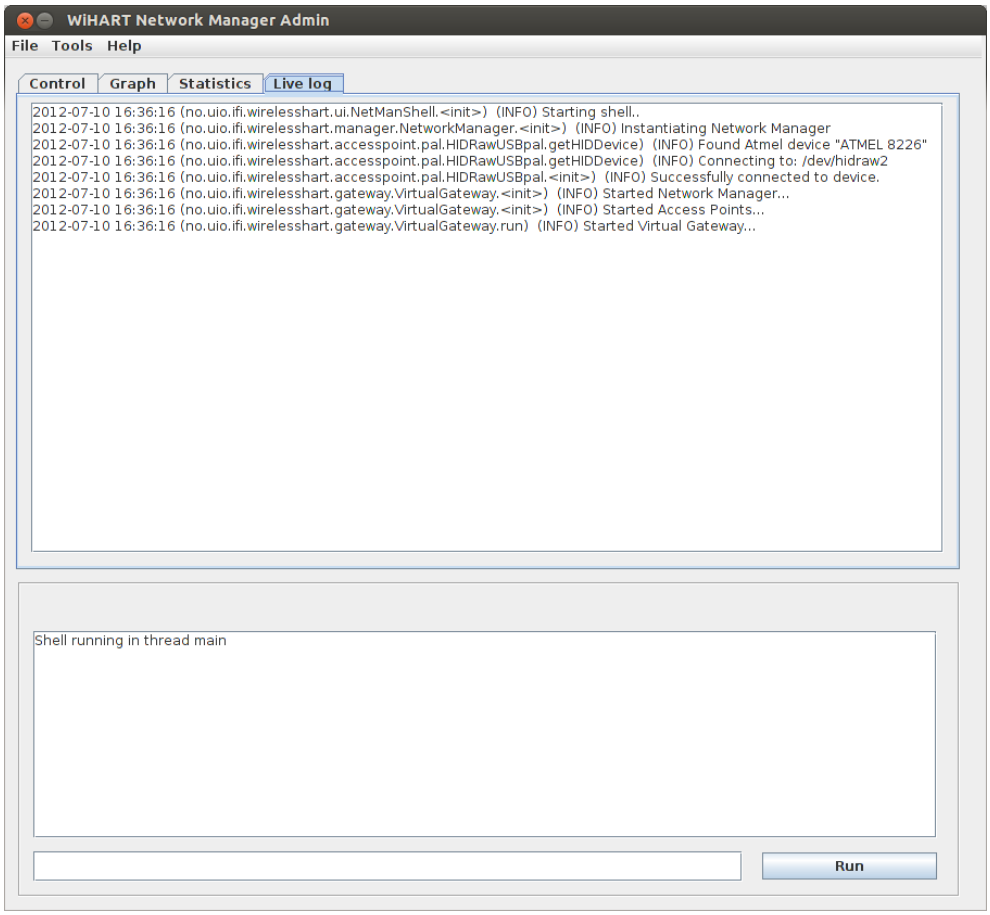


Figure 7.13: The log tab of the GUI

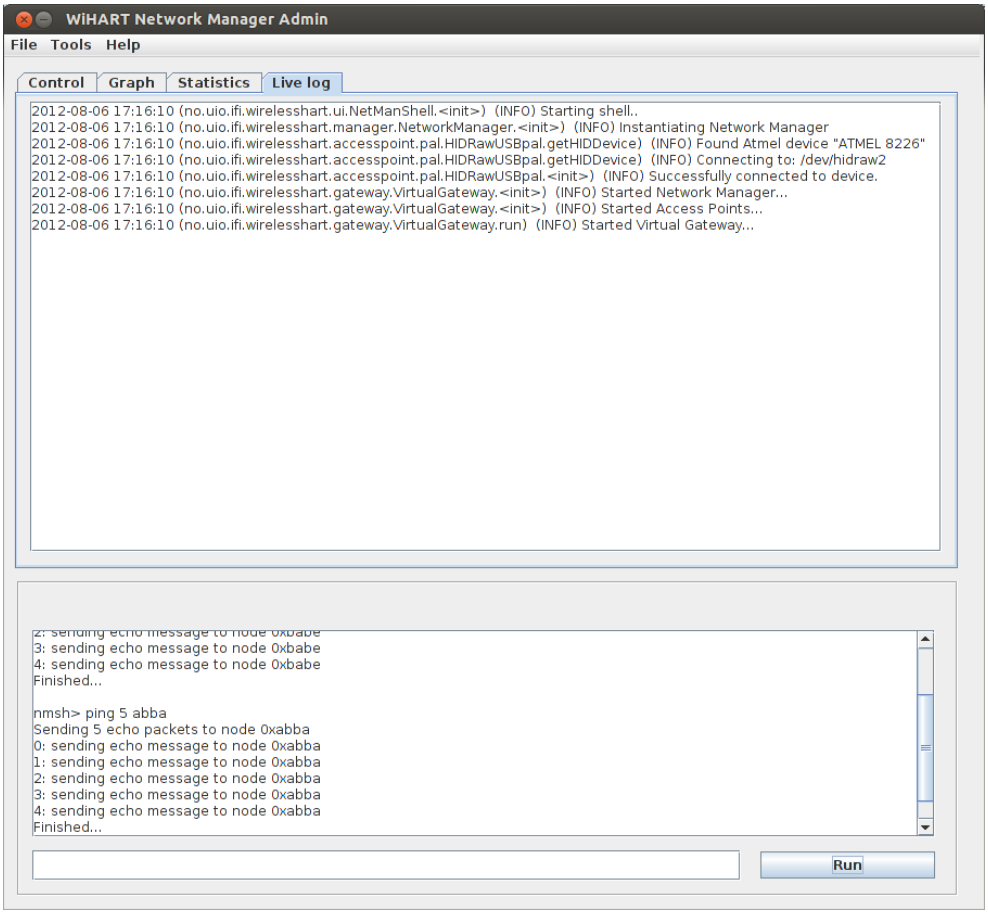


Figure 7.14: An example of the command line usage

Command Line Interface

As GUI development can be tedious and time consuming, and we needed a way to control the behavior of the Network Manager during the run time, we implemented a small command line interface (Figure 7.14) to suit this need. The user interface is simply a `while(true)` loop that blocks and waits for keyboard input. It then matches the command given with a list of functions it has support for, and then runs the given command with the supplied arguments if it is correctly written. This is very helpful when new functions are being tested, as we do not need to statically define behavior in the code. As a consequence of this there is no need for a program restart and recompilation in order to run arbitrary tests.

7.2.3 Security Manager

The standard defines that the relationship between the Security Manager and the Network Manager can be as simple as two classes in the same application, or as separate applications running on different hardware. We have chosen to implement the Security Manager as a class that is instantiated by the Virtual Gateway. The security manager has an API that can be used by either the network Network Manager or the Access Points via the Virtual Gateway. Since the nodes we are using does not support the required encryption type (AES-128 Bit), we did not prioritize to implement the Security Manager, but the correct method definitions and an API for Security Layer payload encryption is implemented. We will document the lack of encryption hardware offloading more extensively in chapter 8.

7.3 Chapter Summary

In this chapter we have provided an overview of the efforts made during implementation, the various approaches taken and the challenges we have encountered on the way. We have given a detailed overview on how the system is currently implemented, starting out with our changes to the implementation of the WirelessHART Field Devices. These modifications and new features include a new architecture for handling TX and RX queues, finishing the implementation of the TDMA state machine and the data link layer join procedures. We have implemented the TDMA state machine into the XMIT and RECV engines on the MAC layer in order to provide a single point of entry into state transitions. In addition, we have provided a description of the functionality for modifying the current system time on the WirelessHART Field Devices in order to keep time synchronized, and the DLPDU construction routines in order to avoid duplicate code.

At the end of this chapter, we have provided a detailed description of our implementation of the WirelessHART Gateway including its subcomponents the Network Manager, Virtual Gateway, Access Points and

Security Manager. We have given a detailed description of our efforts in implementing the Access Point using the Contiki Operating System which we later had to abandon for reasons explained in the text. We have also provided an explanation of which changes we had to perform to the 15dot4-tools project in order to utilize the RavenUSB stick as an Access Point. Following the description of the Access Point implementation is a detailed overview of the implementation of the Network Manager and how it works internally, including the interface for sending HART commands between components, scheduling and serving links and managing it through the implemented Graphical User Interface.

Chapter 8

Testing and Evaluation

In this chapter we provide a walk-through of our test bed set up and configuration in addition to a detailed overview of the network topology of our test network including the node and component names and addresses.

We provide an overview of the various tests performed on both the final system and the tests that were performed under way. We will also evaluate the correctness of the implementation in accordance with the WirelessHART standard.

This chapter will also provide an outline of the topology of the testing network, along with information on how we utilize the hardware available to us to perform traffic dumps and debugging of the network. We then evaluate our implementations, the management of the project described in section 1.6 and how we ended up spending the available time throughout the course of the project.

At the end of the chapter we also provide some information about key issues that were encountered during testing of the code.

8.1 Test Bed Specification

Our test bed is comprised by 2 Linux based computers, two RavenUSB sticks and 4 nodes. One of the Linux computers, named Luigi, is running as a packet sniffer. In terms of hardware it was a Dell Optiplex 280 with

an Intel Pentium 4 3.0GHz CPU and 1GB of ram. This computer ran Debian GNU/Linux 5 and had a connected RavenUSB stick that ran an unaltered version of the Contiki OS. The sniffer used to run Ubuntu 11.10 (Section 4.2.3.1) but this was changed to Debian in order to free up system resources.

The other computer, named Mario, is running Ubuntu 12.04 LTS and was a HP Compaq CZC84802QS with an Intel Core 2 Duo 2.4 GHz CPU and 2GB of ram. Mario is connected to a RavenUSB stick that ran our modified version of 15dot4 tools as well as the Access Point and Gateway application.

The nodes were all running the software specified in our implementation chapter.

8.1.1 Network Topology

In our test network (Figure 8.1), we define 4 nodes in addition to the Gateway and Network Manager. The nodes are all within range of each other and thus, no inter-node routing will be required in the initial setup. This also means that this is a single-hop network and not a multi-hop network eliminating the need for routing as this is not implemented yet. The primary purpose of this test network is to provide a simple and consistent environment in which the data link layer and time synchronization may be debugged, tested and evaluated.

During the initial implementation we disregarded the Gateway and Network Manager and assigned one of the nodes as a time keeping node. This made it possible to have all the other nodes synchronize their clocks to the clock of the time keeping node.

According to the specification [29] the PAN range `0x8000-0x8FFF` is reserved for temporary user defined networks and field trials, so we should utilize for example the PAN id `0x8000` for our network, instead in order for it to be easy to distinguish the PAN id in a packet dump we will use the PAN id `0xCAFE` for our network even if it lies outside the range of temporary networks defined by the standard.

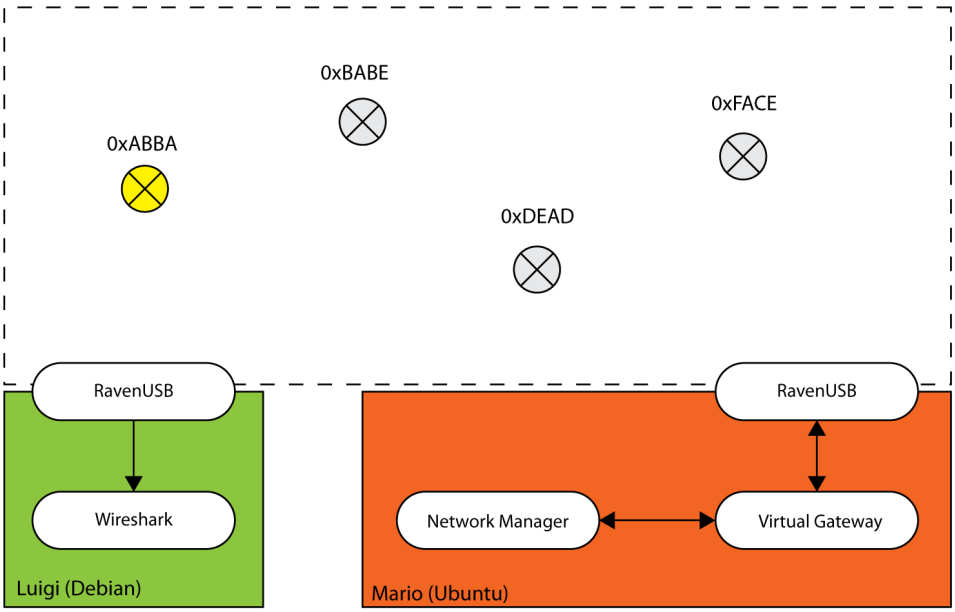


Figure 8.1: The Test bed - This figure shows the layout of our test bed. Luigi is connected through the RavenUSB running the Contiki OS in sniffer mode, sniffing all packets sent by any field device or Access Point and dumping them all to Wireshark which displayed them. Mario is running the Network Manager and Virtual Gateway which was also connected through the RavenUSB, however this one was running 15dot4-tools and worked as an Access Point. `0xABBA`, `0xBABE`, `0xDEAD` and `0xFACE` are the field devices (sensor nodes) where `0xABBA` was the time source node.

8.1.1.1 Nodes

The names of the sensor nodes and their respective short and long addresses are listed in table 8.1. When referring to individual nodes during testing we utilize the short names in this table. For example, we can see that `0xABBA` is designated as a time source while the other nodes will update their internal clocks when a packet from `0xABBA` is received.

Node	Short address	Long address	Time Source
1	0xABBA	0x0123456789ABABBA	Y
2	0xBABE	0x0123456789ABBABE	N
3	0xDEAD	0x0123456789ABDEAD	N
4	0xFACE	0x0123456789ABFACE	N

Table 8.1: Node listing

8.2 Test Results

In the following section we provide an overview of the test performed through the development of the project in order to measure the completeness and accuracy of the implementation. The tests include timing measurements and communication graphs between multiple nodes.

8.2.1 Sending from Primary Node to Multiple Destinations

In figure 8.2 we have configured `0xABBA` as the primary node of the network. "Primary" simply means that it has been statically compiled to act as the only node that is transmitting data DLPDUs. The node has also been configured to send WirelessHART Advertise DLPDUs in any free TX slot at a maximum rate of once per 2 seconds. The generation of data packets is a round-robin to the nodes `0xBABE`, `0xDEAD` and `0xFACE` with 500ms intervals. In addition the superframe is configured so that each of the 3 nodes have

3 links to `0xABBA` within one superframe. Note that in this test we are not using a Network Manager and the superframe has been statically compiled on all the nodes.

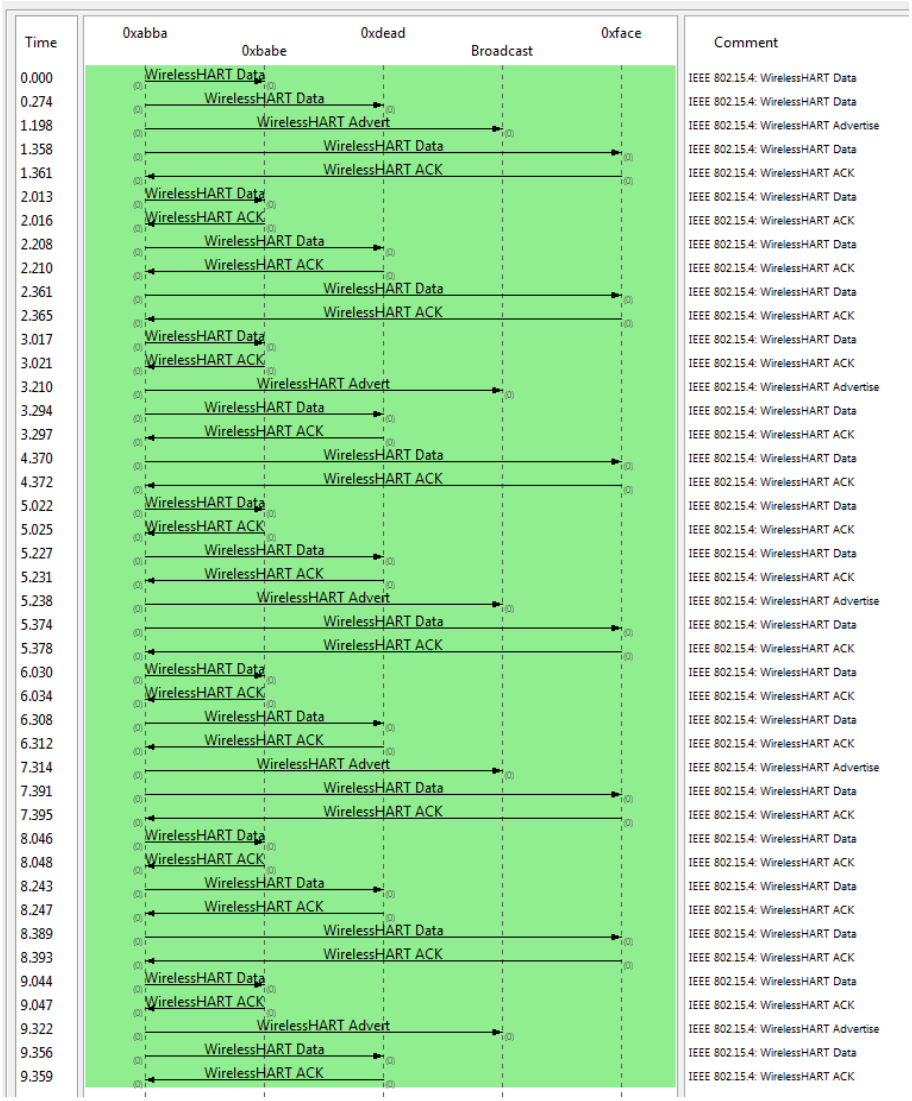


Figure 8.2: `0xABBA` sending data to other nodes - A timing diagram where time is running from top to bottom clearly shows how `0xABBA` is sending data packets to other nodes and receiving acknowledgments from the node which the data packet is directed to. `0xABBA` is also configured to send an advertisement every second which should not receive an acknowledgment.

As we can see by the flow graph, at time 0.000 and 0.274 `0xABBA` is

sending packets to which nobody replies, this is because during this test the other nodes were configured to wait for an advertisement DLPDU before considering themselves officially part of the network. Once the first advertisement DLPDU is broadcasted by `0xABBA` at time 1.198 we can see that the other 3 nodes start sending acknowledgments. We ran this test for 3000 packets and every packet sent by `0xABBA` after the network was formed was properly acknowledged by the destination node. The payload during these tests were a fixed 4-byte integer. The results of this test confirm that the link layer implementation of the TDMA State Machine and the XMIT and RECV engines is functioning correctly.

8.2.2 Single Node Timing Accuracy

During testing we had some problems with the timing. While the sensor node claims it is sending out packets every second with an inaccuracy of a few hundred microseconds, the sniffer (Luigi) using the RavenUSB stick claims to only receive a packet every 1.012 seconds which is a difference of approximately 12 milliseconds. While we do not yet completely understand where this delay is coming from, after some debugging and tracing we believe it is due to the clock crystal on the sensor nodes. If the number of ticks required for a second is any less or a bit more inaccurate than on the PC these two systems will disagree on the duration of one second. We confirmed our theory by modifying the number of tick needed to count a second on the sensor node, and observed that the difference between the systems became much smaller.

8.2.3 Multiple Node Timing Accuracy

Another problem arose during initial testing of multiple active nodes at the same time. The sniffer (Luigi) registered each packet with a timestamp relative to the previous packet received from any node. We configured two nodes to send a packet every second unconditionally, with no acknowledgments, no processing, etc. In this setup we observed that the clock skew between the nodes was linearly increasing. If the nodes stared out sending

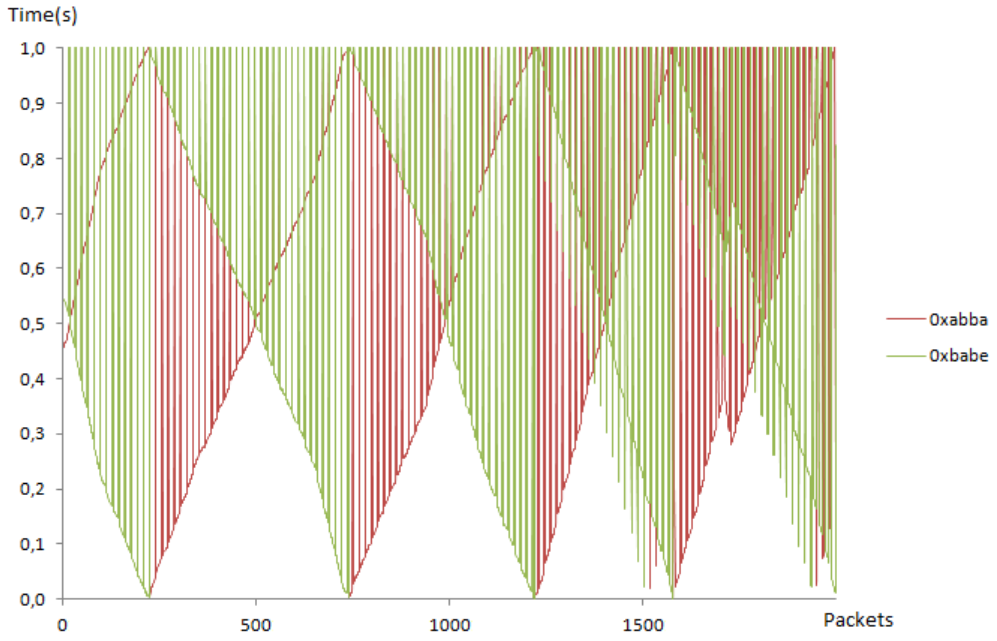


Figure 8.3: Timing diagram - Showing how two nodes which are configured to send one packet every second completely independently of each other are slightly different incurring an increasing clock skew between the nodes. The plot illustrates that the clocks of these two node have different notions of time.

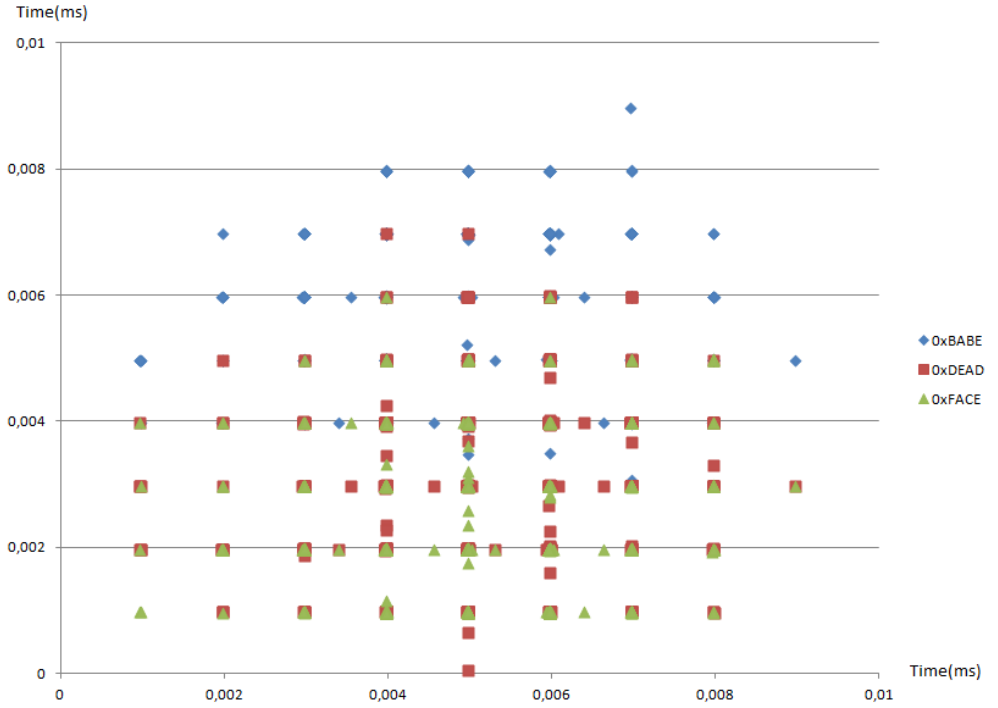


Figure 8.4: Timing diagram - Showing how 0xBABE, 0xDEAD and 0xFACE drift in relation to 0xABBA. The closer the data points are to the origin the more accurate the timing is relative to 0xABBA. Note that both axes show the same values, this is to create a plot in which we can measure accuracy in groups of data points close to origin. We can see by this plot that the green data points (0xFACE) is the node that skews the least relative to the time source (0xABBA) and 0xBABE displays the most amount of skew.

with 0.5 seconds offset to each other, after only 200 seconds the nodes had drifted enough to be sending directly on top of each other. We initiated a test of 2000 packets for each node, sent every second and the plotted result can be seen in 8.3. From the graph we can see that after only just below 500 seconds the nodes have wrapped around, giving us an average skew between the nodes of about 2ms.

In the second plot (Figure 8.4) We observe the relative time difference between the 0xABBA and the other nodes. Note that both the X and Y axis plot the same values. The closer the data points are to the origin

and the tighter the groups are, the smaller relative time differences. As we observe in the plot, especially with 0xBABE we experience a rather large variation in timing differences while 0xFACE is the node that is most accurate in relation to 0xABBA leading us to believe that the clocks on the micro controllers are running on slightly different frequencies or that these variations are caused by tiny differences in the crystals.

8.3 Evaluation of Data Link Layer Implementation

The implementation of the Data Link Layer on the WirelessHART Field Devices have through the project gained several new features. The implementation of the transmit and receive queues has been simplified in order to reduce the complexity of access and provide a cleaner interface for the XMIT and RECV engines. This in turn simplifies the algorithms and makes it simpler to confirm that the code works as expected, which it does to a satisfactory degree.

The basic primitives for time synchronization have been implemented including the functionality for adjusting the system time on the WirelessHART Field Devices. We have been able to confirm that changing the system time works, but not if there are any adverse effects to timers already running. An alternative option to adjusting the internal time on the nodes would have been to implement a temporary variable to hold the difference between the actual system time and the WirelessHART ASN. We haven chosen to implement the first approach as we think it is cleaner and more architecturally correct.

A common entry point for the TDMA state machine has been implemented and the implementation of the XMIT and RECV engines have been re-factored to use this entry point for inducing state transitions. We believe that this creates more separation between the TDMA state machine and the other components of the Data Link Layer. In addition, having a single point where the radio transitions into RX and TX mode will makes

it easier to implement channel hopping in the future.

We have also implemented the basic functionality for construction on Data Link Protocol Data Units and this will serve as a starting point for future implementations of other packet types. At this point we have implemented support for the DATA and ADVERTISE packet types and further packet types may be added quite easily by extending the current implementation.

Through the tests performed in section 8.2 we have confirmed that the nodes function correctly in terms of sending and receiving packets, in addition to handling the packet buffers. The nodes are able to sustain operation over an extended period of time and respond to packets directed to them. We have had issues when putting the radio into sleep mode but have chosen to not prioritize debugging of these issues as the primary focus has been on implementing the features described in section 6.5 and we consider sleeping the radio when IDLE as a secondary objection.

8.4 Evaluation of Gateway Implementation

This section provides an evaluation of the Gateway that has been implemented during the course of this project.

8.4.1 Access Point

In order for the Gateway to be able to communicate with the field devices it needs Access Points (APs). When we started implementing the Gateway and Network Manager it was obvious we needed functioning APs as soon as possible. Since the RavenUSB has the same radio as the Raven boards used for the WirelessHART Field Devices, it is easily connected with the computer running the Network Manager through the USB port and can also use the computer as its power source it seemed like the best choice of our available hardware for running the APs.

When researching suitable software for the RavenUSB sticks we came over the Contiki OS first and decided to try it as sniffer software and later

for the APs as well. The Contiki OS has performed very well as a sniffer together with Wireshark compared to the Daintree sniffer which would not capture all packets. As described earlier using Contiki for the APs was less successful and a lot of time was spent trying to adapt it to WirelessHART which in the end was too time consuming and we decided to abandon it. Looking back at this we could have looked for more alternatives other than Contiki for the APs from the beginning and thus have saved time by changing to 15dot4-tools earlier in the project.

15dot4-tools was a lot smaller than Contiki and did not assume the 802.15.4 SICSLowPAN standard, instead it looked at all the data being sent and received as raw 802.15.4 data. This allowed us to use the RavenUSB sticks with 15dot4-tools as the physical layer of the APs, where the APs upper layers located on the host machine would handle the WirelessHART parts of the packet and 15dot4-tools would just send the data it received from the upper layers. The same applies when receiving packets; the RavenUSB running 15dot4-tools will receive data and send it to the host machine through the JavaHID API and make no assumptions to what type of data it is. The APs work as expected and are able to both send and receive data correctly, i.e. the data sent is identical to the data being received. 15dot4-tools is also able to change the radio's channel on request by the APs higher levels so it is ready for channel hopping in the future.

Having all of the APs layers, except for the physical layer, on the host machine gives us several advantages. First of all, the rest of the Gateway is implemented in Java, so having most of the functionality of the APs in Java makes communication between the devices simpler as the RavenUSB runs C code. Another advantage is that there is less communication with the RavenUSB since communication between the host and the USB stick only takes place when a packet is being sent or the AP expects a packet to be received. Lastly, based on our experience the implementation time in Java is usually shorter than in C, especially with larger projects. This meant that we were likely to get more done on the APs when programming in Java than if we would have started to write all functionality in C on the RavenUSB and integrating it with the 15dot4-tools.

8.4.2 Virtual Gateway and Network Manager

After implementing the Access Point and ensuring that it was communicating with the nodes unrestricted we found that we had to respect the logical separations of the Gateway elements. We first created the Gateway class which starts the Network Manager and the Virtual Gateway. The Virtual Gateway creates and puts all Access Points in a list and creates an RX queue and a TX queue for each Access Point. We found that the Access Points should be as light-weight as possible, and also be able to run on systems with limited resources. The reason for the API was because the standard depicts that an Access Point should be able to communicate with the Gateway using i.e. a TCP/IP connection. Although large alterations to the Access Point code have to be changed to accommodate this, we chose to keep the logical separation as the standard described. As an added bonus this also worked well as a way of sectioning the code within the project.

After the Network Manager is instantiated by the Virtual Gateway, it essentially creates a superframe and sets the links to its neighbors. As of now this is statically set, but this can, based on our implementation easily be extended to be set on the basis of joins and part messages from nodes. The Network Manager is fairly simple, and does not support any functionality apart from keeping track of links, neighbors and super frames for the network at hand. However, the logical separation of the Network Manager is thought to work well in the future when Network Layer functionality is added, i.e. routing and multi-hop links.

8.4.3 Graphical User Interface

We chose to implement the Graphical User Interface (GUI) because the amount of information we had to see simultaneously was becoming too vast. We also wanted a way to interact with the application during runtime, not only defining the behavior statically in the program code.

This led to the development of a simple GUI based on Java Swing. The first “module” we implemented was a statistics pane that retrieves

statistics-objects from the Virtual Gateway and the Access Points, and shows this dynamically. This was a great improvement to printing it out to the console. But this still has the potential to show more performance metrics that can be useful for debugging. The second element we implemented was the shell. This enabled us to interact with a running Virtual Gateway, in order to send the desired kind of packets and review the results during run-time. The shell does not have very many commands implemented, and the communication with the network is only one-way. Meaning that the feedback is printed to the standard output (stdout). But this improved the efficiency of debugging, as we did not need to restart the application to apply minor changes.

We also wanted to see the network "real-time" so we chose to input the graph pane and a tree pane. The graph pane is dynamically updated every N second and shows the network as is. This graph also takes into consideration multi-hop scenarios. This because we thought it would be helpful to be able to see which device is neighboring who, and also to see this change if the devices are physically moved.

After doing this we also thought that it would be useful to see device settings real-time as well. We then proceeded to implement a tree pane that was built from the current superframe, and added the Network Manager and the Virtual Gateway. We did not get as far as filling this up with useful info about state variables and such. But with this functionality, it will be easier to see what settings are active on the devices at any point in time. And this is arguably better than printing these values to the standard output. Especially if the network is extended to include more nodes.

Lastly we implemented the logger to a separate window in the GUI, this was done to strip the program console output to a bare minimum. Each class has its own logger object that prefixes log entries accordingly and prints it to this GUI window. This helped clean up the console output, but can be used to a larger extent in order to improve trace-ability of program outputs.

8.5 Evaluation of Project Management

We started out by getting to know the WirelessHART specification separate to the code base in order to get an overview of the standard and form an idea about what the project was all about. After spending approximately a month learning about the specification and playing with the hardware we moved on to test and evaluate the already existing implementation as described in section 1.5. We did run into a few issues that needed to be fixed before moving on to extending the implementation and these issues are more thoroughly described in section 5.6.

After fixing these issues we were able to move on and extend the link layer further by implementing the TDMA state machine to allow proper transitioning between the various states described by WirelessHART.

The biggest time consumer throughout this project has been the design and implementation of the Gateway and Network Manager. We have spent a lot of time trying multiple approaches for implementing an Access Point and getting the computer on which we have been running the Gateway to communicate with the wireless network.

Testing and evaluation of the implementation both on the wireless nodes and on the Gateway has been a continuous process which has intensified towards the end of the project. An overview of the time spent on the different tasks by the authors can be seen in figure 8.5. In regard to overall time managements we have stayed fairly close to the time division illustrated in figure D.1 although the time spent on implementation is higher than the time spent on design towards the end of the project.

8.6 Problems Encountered

This section provides information on different problems that have been encountered throughout the project and how we tried to solve them.

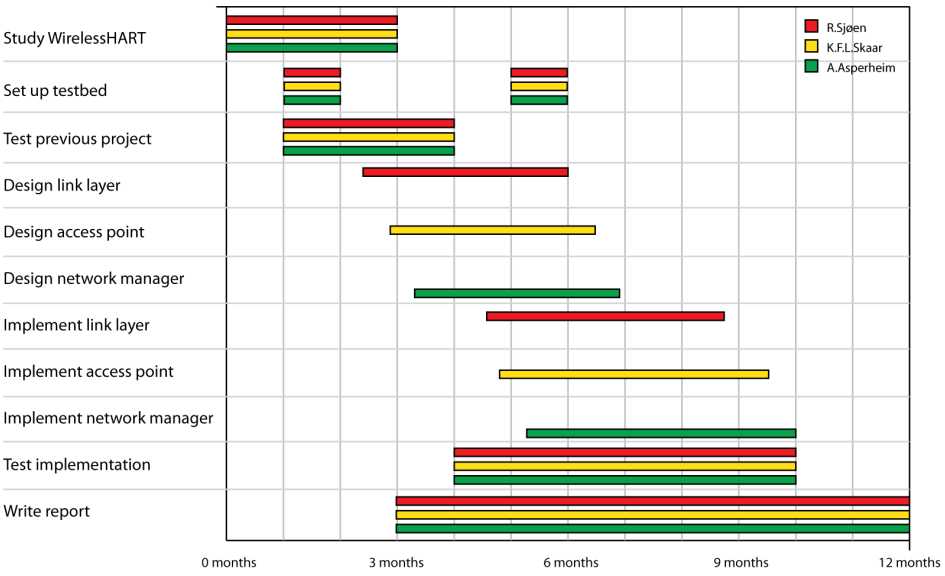


Figure 8.5: Project Management - Overview of Time Spent. Each line represents the approximate amount of time spent by the individual on a certain task. The design and implementation of the Link Layer, Access Point and Network Manager were mainly managed by Sjøen, Skaar and Asperheim respectively but not exclusively. These parts, however, were often shared amongst the authors when it became apparent that one had more to contribute with at that specific task.

8.6.1 Node to Node Timing Accuracy

Inter-node timing and synchronization has been one of the primary focus areas during implementation. Debugging timing issues has proven to be very difficult as attaching the debugger to one of the nodes while running will significantly decrease the clock speed of that node causing the node to operate with a different notion of time than the other nodes. The timing works to a certain degree, nodes are able to roughly synchronize their ASN upon receipt of an advertisement packet and correctly listen to TX and RX slots when packets should be sent or transmitted. However, there are issues when putting the node to sleep in between served slots and so far we have not been able to make this work.

8.6.2 Gateway to Field Device Timing Accuracy

Timing requirements between the WirelessHART Field Devices and the Access Point of the Gateway has also proven difficult to debug and test. While the Gateway itself believes it is sending the correct packet at the correct time, including advertisements with synchronization data like the node to node timing, the node seems unable to synchronize as accurately as when working against another node.

8.6.3 Timing Accuracy on the Gateway Computer

As a consequence of the timing issues on the nodes we wanted to investigate whether the timing on our Network Manager and Access Point were reliable. The way we did this was by sending a query to the NTP (Network Time Protocol) server timekeeper.uio.no in a script over 5 hours and checking the difference. The difference was plotted in the figure 8.6 which shows that 93% of the samples lay within the allowed deviation of 100μs described by the WirelessHART standard. We had outliers that would undoubtedly break the slot scheme in a superframe. For testing at this stage we decided that this was sufficient. For other applications an operating system which has a stricter control of the kernel preemption should

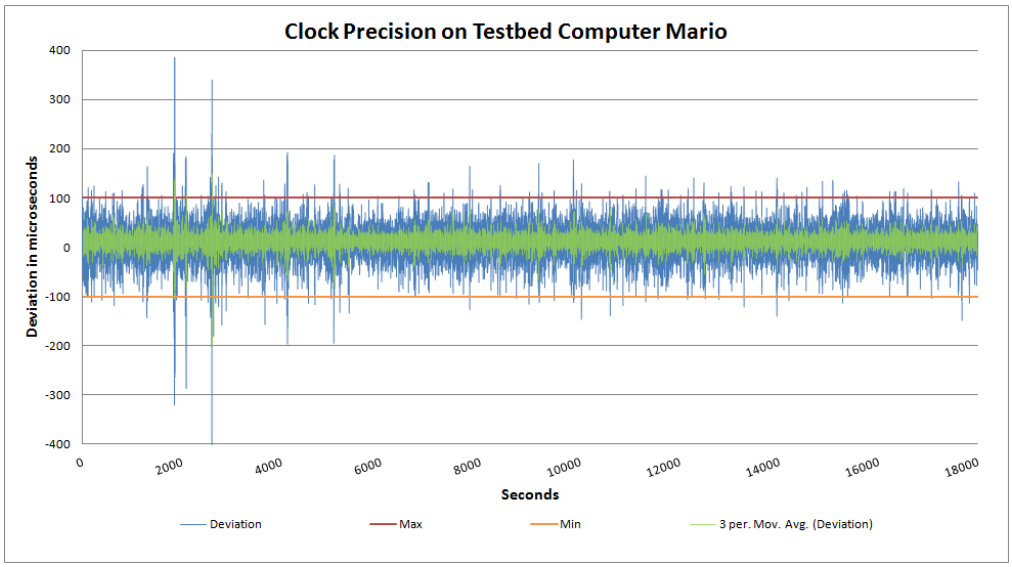


Figure 8.6: The Y Axis shows the time in microseconds and the X axis shows seconds from when the test was started. The picture shows how much offset the system clock had at the point of measure against the NTP (Network Time Protocol) server timekeeper.uio.no. The green field shows a trend-line that samples an average over 3 seconds (the length of a superframe).

be considered, and the Real Time Linux project [12] may be an interesting alternative.

8.6.4 Lack of Hardware Encryption Support

Since the standard depicts that the security layer payload is to be ciphered with an AES-128 bit encryption, it is imperative that the hardware supports this. Since this is not a feature of the ATmega1284P, and we found no good reasons for software implement this when other and newer micro controllers has this on-chip. This adds to why this micro controller is unsuited for a complete implementation of the Wireless HART stack. The ATmega128RFA1 board is one example of a RF capable device that supports this on-chip. So to work around this problem, further work should be on other hardware.

8.6.5 Enabling Short Address Mode in Contiki

By default the Contiki OS uses the long address mode (8 bytes) when sending packets and assumes that packets received also use the long address mode. We mostly use short address mode (2 bytes), also called nickname, when testing sending packets through the WirelessHART network. In order to use Contiki we had to enable the short address mode so it could communicate with the field devices using nicknames in the source and destination fields in the DLPDU. Fortunately the Contiki OS has support for both short and long address modes and in order to enable nicknames we only had to set the default to short in stead of long. This has been described in more detail in section 7.2.1.1.

8.6.6 Fixing the Payload in Contiki

When sending packets using the Python test program and the RavenUSB running Contiki we noticed that when the packets showed up in Wireshark the payload part of the DLPDU had changed. The first few bytes of the payload had been overwritten by something else. We went through the code trying to find out where exactly these bytes where coming from, but after spending a significant amount of time on it we decided to just offset the payload so it started after where it was overwritten. This was explained in more detail in section 7.2.1.1.

8.6.7 Setting the DLPDU Specifier in Contiki

The DLPDU specifier, as explained earlier in section 3.3.5, gives information of the packets priority and type which is important for further processing of the packet. When sending a packet the DLPDU specifier field of the DLPDU would be overwritten by something else by Contiki and the DLPDU specifier byte would just disappear. We tried finding out where Contiki would edit the data we sent it, but the amount of code in Contiki made this a huge task we did not have time for. We were able to set the DLPDU specifier statically in Contiki, which was explained further

in section 7.2.1.1.

8.7 Chapter Summary

In this chapter we have outlined the testing environment and the problems encountered from the perspective of the tester, we have also provided a detailed overview of the tests performed on the implemented software in order to verify its correctness and evaluate our progress. We have outlined any shortcomings which have been exposed as a result of this testing process. A large part of this process is making sure that the implementation actually performs as we expect it to.

Chapter 9

Conclusion and Future Work

The problem definition from section 1.2 states the following:

> The overall goal of this project is to implement the minimum requirements for a working WirelessHART [29] network running on the hardware provided by Atmel in addition to a functioning Network Manager.

In this chapter we will provide an evaluation of the project as a whole, which conclusions we have drawn from working with the project and a road map for future work.

9.1 Conclusion

During the course of the project we have described the various protocols for wireless sensor networks that exist and provided a detailed outline regarding the implementation of the WirelessHART standard.

In addition to this we have also accomplished to continue development on this project and have successfully implemented several additional key elements in the WirelessHART protocol stack.

These elements include proper link management on the MAC layer, we have optimized the link selection routines, the packet creation routines and the management of TX and RX queues. We have also provided a coarse-grained implementation and design of the network layer.

The XMIT and RECV engines on the MAC layer have been re-factored and some bugs have been fixed. The receiving and processing of advertisements in the MAC layer have been implemented allowing multiple nodes to stay in synchronization and communicate with each other over an extended period of time. We have verified the already existing implementation of the link scheduling algorithm and find that it is working as expected.

Looking at the results we have to a satisfactory degree met the functional and non-functional requirements defined in section 6.2 and 6.3 by priority. The code written is well commented where needed and function and variable naming is logical based on their context and usage. This makes the code base easier to understand and maintain.

When looking back at the problem definition for our thesis, which has been restated at the beginning of this chapter, we conclude that the goal we set was partially achieved, a more detailed description of which part of the project we consider as completed will follow in the next section.

9.1.1 Meeting Requirements

In order to conclude what has been achieved during this project we need to look at the requirements listed in sections 6.2 and 6.3 and conclude whether these requirements have been met or not.

9.1.1.1 Functional Requirements

In this section we will look at each functional requirement listed in section 6.2 in detail and conclude whether or not it has been met to a satisfactory level.

A Working XMIT and RECV Engine on the Link Layer (FR01)

The implementation of a XMIT and RECV engine is almost complete and function as expectedly, we have had no serious problems with the implementation of these. The XMIT engine is able to successfully build and transmit a packet retrieved from the TX queue

of the link being served. The RECV engine is able to successfully receive a packet and unpack it to pass the payload up to a higher layer, sending an ACK back to the source should that be required.

Service Primitives for Local Management (FR02)

The interface for the service primitives on the Data Link Layer has been implemented and the implementation needs to be continued in order to provide a full implementation of the Data Link Layer. This is also mentioned in future work (Section 9.2).

Multiple RX and TX Queues (FR03)

The basic architecture of the Data Link Layer and the handling of the RX and TX queues have been rewritten in order to support a global RX queue and a separate TX queue for each neighbor. This simplifies the implementation quite a bit and there are less data structures to keep track of.

Interface Between Computer and Sensor Nodes (FR04)

In order for the Gateway to communicate with the sensor nodes from the host computer an interface was needed. This interface consists of the Access Points (APs) communicating via the JavaHIDAPI with the RavenUSB sticks running 15dot4-tools. By being able to interface the RavenUSB sticks we can now successfully communicate with the rest of the WirelessHART network from our host computer running the Gateway. Although this might seem like a trivial task it took a lot of work to accomplish this, starting with a failed attempt of using the Contiki OS, abandoning it and then starting over with 15dot4-tools. Looking back at the evaluation of the AP implementation in section 8.4.1 we can with confidence say that we have fulfilled this functional requirement.

Time Synchronization (FR05)

Time synchronization has been partially implemented on the WirelessHART Field Devices. Functionality for adjusting the internal clock has been implemented and a coarse synchronization upon the

reception of a WirelessHART Advertisement is working. This work needs to be continued to continuously adjust the clock when the node receives a packet from any source which is marked as a time source. This is further discussed in future work (Section 9.2).

Network Manager (FR06)

In order to implement a Network Manager we also had to implement a Gateway and a Virtual Gateway. The Network Manager is implemented and in use, but with limited functionality. The data structures and layout of the Virtual Gateway and Network Manager are scalable and offers a baseline for further development. Most of the work done to the Gateway is done in the Virtual Gateway and the Access Point modules, and these function well in the current setup and will also offer a scalable and flexible base for future work.

9.1.1.2 Non-Functional Requirements

Below is a conclusion of whether or not each of the non-functional requirements listed in section 6.3 have been met or not.

Processing of a Specific Slot Within Timing Limitations (NFR01)

Throughout the testing and evaluation of the implementation we have been able to confirm that a slot is properly served within timing limitations leaving this requirement as completed. As a result, due to the efficiency of the link scheduler in selecting and scheduling a new link at the end of each slot this leaves a significant margin of error available for slot timing.

Proper Documentation of API (NFR02)

API Documentation is provided by appendix F in addition to documentation generated by JavaDoc and Doxygen for the WirelessHART Gateway and the WirelessHART Field Devices respectively. We consider this requirement completed.

Documentation of Non-API Functions (NFR03)

Non-API functions have been documented through comments in the code where necessary for readability. In addition, a detailed set of reference documentation which includes non-public functions have as explained in the previous section been generated for both the WirelessHART Gateway using JavaDoc, and the WirelessHART Field Devices using Doxygen.

Low Power Consumption, Minimal use of Radio (NFR04)

This requirement has not been implemented apart from the final implementation of the TDMA state machine described in section 7.1.2. These changes make it simple to disable the radio when the state is IDLE, but further debugging needs to take place to confirm timing limitations in the hardware devices. This requirement has not been completed and has been included in the future work section (Section 9.2).

Separation of Layers (NFR05)

Throughout the project we have worked on building a clearer separation between the layers of the WirelessHART protocol stack, especially the physical layer and the link layer in order to sever the connection between the transceiver and the rest of the physical layer. This is important in order to be able to move the implementation onto a different hardware device in the future. While we have been able to separate some parts of the layers this requirement has not been fully completed and has been included in the future work section (Section 9.2).

9.1.2 Hardware

Looking back at our experiences with the hardware we used for our Field Devices (Raven boards) and Access Points (RavenUSB sticks) we can see there were some issues. The lack of hardware offloaded encryption support, the inaccuracy of the RC Oscillator on the nodes and the issues

with changing the clock source are the main problems with this hardware. These issues make it impossible to meet the requirements for timing and encryption, set by the WirelessHART standard[29], with the current hardware. We therefore conclude that the current hardware is not suitable for a full-fledged WirelessHART network and, for instance, the newer ATmega128RFA1 (Section 4.1.2) with a single-chip solution and support for AES should be considered as a new hardware option.

9.2 Future Work

We have identified a few key areas in which we believe future efforts are required in order to provide a fully functioning implementation of the WirelessHART standard, in this section we will describe each of these areas in order to provide a starting point for future work.

9.2.1 Design and Implementation of Network Abstraction Layer

We have only partially designed and implemented the Network Abstraction Layer (NAL). This has been done on purpose in order to provide a good starting point for implementation of this layer. The service primitives on the NAL need to be identified and the functionality needs to be implemented.

9.2.2 Debug and Fix Timing Between Gateway Access Points and Sensor Nodes

During testing, there were issues with staying synchronized between the sensor nodes and the WirelessHART Gateway. We believe this is mainly due to the nodes and computer having a different notion of time and this needs to be debugged further.

9.2.3 Implement Channel Hopping and Blacklisting

In order to have a fully functional WirelessHART MAC layer channel hopping and blacklisting need to be implemented. The implementation of this functionality will be fairly easy as it only includes implementing a pseduerandom generator for channel selection in accordance to the WirelessHART standard.

9.2.4 Implement Routing

Our test network during this project has been a single-hop network. In order to provide a fully functional WirelessHART network layer routing needs to be implemented in the Gateway and the nodes need to be able to receive a superframe from the Network Manager and respect the information contained in it. A good routing algorithm for sensor networks also needs to be identified and implemented on the Network Manager.

9.2.5 Upgrading Hardware

Seeing that the hardware we are currently using seems to be unfit for the WirelessHART timing and encryption requirements, we suggest that further development should be based on different hardware.

Bibliography

[1] 15dot4 tools. http://www.newae.com/tiki-index.php?page=15dot4tools. [Online] Accessed March 2012.

[2] The contiki operating system. http://www.contiki-os.org/. [Online] Accessed February 2012.

[3] Graphviz - graph visualization software. http://www.graphviz.org/. [Online] Accessed March 2012.

[4] Hart communication foundation. http://www.hartcomm.org/. [Online] Accessed March 2011.

[5] Ieee 802.11(tm): Wireless local area networks (LANs). http://standards.ieee.org/about/get/802/802.11.html. [Online] Accessed February 2012.

[6] Java api for working with human interface usb devices (hid). http://javahidapi.googlecode.com/. [Online] Accessed March 2012.

[7] Lint4j - lint for java. http://www.jutils.com/. [Online] Accessed April 2012.

[8] Lua - a script language for rapid protoyping used in wireshark. http://wiki.wireshark.org/Lua/. [Online] Accessed February 2012.

[9] Miwi wireless networking protocol stack. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en520606. [Online] Accessed November 2011.

[10] Ndiswrapper.

[11] Ohloh - contiki os project summary. https://www.ohloh.net/p/8645. [Online] Accessed February 2012.

[12] Real time linux foundation. http://www.realtimelinuxfoundation.org/. [Online] Accessed July 2012.

[13] Roll work group. http://wiki.tools.ietf.org/wg/roll/. [Online] Accessed February 2012.

[14] Sam7s-ek. http://www.atmel.com/tools/SAM7S-EK.aspx/. [Online] Accessed June 2012.

[15] Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs). http://www.ieee802.org/15/pub/TG4.html. [Online] Accessed March 2012.

[16] Wireless systems for industrial automation: Process control and related applications. http://www.isa.org/ISA100-11a. [Online] Accessed September 2011.

[17] Wirelesshart project website. http://folk.uio.no/andeas/whart/. [Online] Accessed March 2012.

[18] Wireshark - an open source network analyzer. http://www.wireshark.org. [Online] Accessed February 2012.

[19] Zigbee specification. http://www.zigbee.org/Specifications/ZigBee/Overview.aspx. [Online] Accessed September 2011.

[20] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans). *IEEE Std 802.15.4-2006*, 2006.

[21] Norman Abramson. The aloha system: another alternative for computer communications. In *Proceedings of the November 17-19, 1970,*

fall joint computer conference, AFIPS '70 (Fall), pages 281–285, New York, NY, USA, 1970. ACM.

[22] Juan Hector Sanches Alvarez. WirelessHART network manager design. Master's thesis, Royal Institute of Technology, May 2011.

[23] Atmel. *AVR2025: IEEE 802.15.4 MAC Software Package User Guide*. 2025-AVR-04/09.

[24] Atmel Corperation. Avr068: Stk500 communication protocol. *Atmel*, 2006.

[25] M. Felser. *Profibus Manual*. epubli.

[26] Rong gang Bai, Yu gui Qu, Yang Guo, and Bao hua Zhao. An energy-efficient tdma mac for wireless sensor networks. In *Asia-Pacific Service Computing Conference, The 2nd IEEE*, pages 69 –74, dec. 2007.

[27] Miaomiao Hua and Lida Dong. A closed-loop adjusting strategy for wireless hart time synchronization. In *Communications and Information Technologies (ISCIT), 2011 11th International Symposium on*, pages 131 –135, oct. 2011.

[28] J.W. Hui and D.E. Culler. Ipv6 in low-power wireless networks. *Proceedings of the IEEE*, 98(11):1865 –1878, 2010.

[29] IEC. *Industrial communication networks - Fieldbus specifications - WirelessHART(tm) communication network and communication profile*, 1.0 edition, 01 2009. IEC/PAS 62591.

[30] ISO/IEC. *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*, 1994. ISO/IEC 7498-1:1994.

[31] A.N. Kim, F. Hekland, S. Petersen, and P. Doyle. When hart goes wireless: Understanding and implementing the wirelesshart standard. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 899 –907, sept. 2008.

[32] N.P. Mahalik. *Fieldbus Technology: Industrial Network Standards for Real-Time Distributed Control*. Engineering Online Library. Springer, 2003.

[33] Hyung G. Myung, Junsung Lim, and David J. Goodman. Single carrier fdma for uplink wireless transmission. *Vehicular Technology Magazine, IEEE*, 1(3):30 –38, sept. 2006.

[34] Oracle. Javadoc: Java incode documentation.

[35] Codeminers Java HID API Project. Java api for working with human interface usb devices (hid). Online, 2012.

[36] E.M. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *Personal Communications, IEEE*, 6(2):46 –55, apr 1999.

[37] D. Saha and T.G. Birdsall. Quadrature-quadrature phase-shift keying. *Communications, IEEE Transactions on*, 37(5):437 –448, may 1989.

[38] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010.

[39] Jianping Song, Song Han, A.K. Mok, Deji Chen, M. Lucas, and M. Nixon. Wirelesshart: Applying wireless technology in real-time industrial process control. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 377 –386, april 2008.

[40] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[41] Håvard Tegelsrud and Jørgen Frøysadal. WirelessHART - review and implementation. Master's thesis, University of Oslo, Institute of Informatics, 2010.

[42] Mangalarapu Chaitanya Kumar Thamer Alyass. Design & implementation of time synchronization for real-time wirelesshart network. Master's thesis, Jönköping University, 2010.

[43] International Telecommunication Union. *Radio Regulations*. 4th edition.

[44] Dimitri van Heesch. Doxygen: Source code documentation generator tool, 2008.

[45] Andrew J. Viterbi. *CDMA: principles of spread spectrum communication*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

List of Figures

List of Tables

Appendix A

Definitions

ACK Acknowledgment

AES Advanced Encryption Standard

AF Address Family

API Application Programming Interface

AP Access Point

ASN Absolute Slot Number

BMM Buffer Management

CCA Clear Channel Assessment

CDMA Collision Detect Multiple Access

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

CTS Clear To Send

DLL Data Link Layer

DLPDU Data Link Protocol Data Unit

DSSS Direct-sequence spread spectrum

EEPROM Electrically Erasable Programmable Read-Only Memory

EUI Equipment Unique Identifier

FDD Frequency Division Duplexing

FDMA Frequency Division Multiple Access

FFD Full Function Device

FIFO First In First Out

FTSP Flooding Time Synchronization Protocol

GPIO General Purpose Input/Output

GSM Global System for Mobile

GUI Graphical User Interface

GW Gateway

HART Highway Addressable Remote Transducer

HCF Hart Communication Foundation

HID Human Interface Device

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IRQ Interrupt Request

ISA International Society of Automation

ISM Industrial, Scientific and Medical

ISO International Organization for Standardization

ISP In System Programming

JDK Java Development Kit

JNI Java Native Interface

JRE Java Runtime Environment

JTAG Joint Test Action Group

LAN Local Area Network

LCD Liquid Crystal Display

LLC Logical Link Control

LLN Low power and Lossy Networks

LTS Long Time Support

MACA Multiple Access with Collision Avoidance

MAC Medium Access Control

MCL MAC Core Layer

MCU MicroController Unit

MIC Message Integrity Check

MTU Maximum Transmission Unit

NAL Network Abstraction Layer

NDIS Network Driver Interface Specification

NIC Network Interface Controller

NL Network Layer

NM Network Manager

NPDU Network Protocol Data Unit

NTP Network Time Protocol

OSI Open Systems Interconnection

OS Operating System

PAL Physical Abstraction Layer

PAN Personal Area Network

PDU Protocol Data Unit

PF Packet Family

PHY Physical

PIB PAN Information Base

PID Product ID

PPDU Physical Protocol Data Unit

QMM Queue ManageMent

QPSK Quadrature Phase Shift Keying

RAM Random Access Memory

RCM RPL Control Message

RC Resistor Capacitor

RECV Receive

RFD Reduced Function Device

RF Radio Frequency

RNDIS Remote Network Driver Interface Specification

ROLL Routing Over Low-power and Lossy networks

ROM Read-Only Memory

RPL Routing Protocol for LLN

RTC Real Time Clock

RTS Request To Send

RX Receive

SIO Serial I/O

SNA Sensor Network Analyzer

SPDU Security Protocol Data Unit

SP Service Primitive

SRAM Static Random-Access Memory

TAL Transceiver Abstraction Layer

TDD Time Division Duplex

TDMA Time Division Multiple Access

TFA Transceiver Feature Access

TSCH Time Synchronization and Channel Hopping

TTL Time To Live

TW Two Wire

TX Transmit

UP Unified Process

USB Universal Serial Bus

UTF Unicode Transformation Format

VG Virtual Gateway

VID Vendor ID

VPN Virtual Private Network

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

XMIT Transmit

XTAL Crystal Oscillator

Appendix B

PAN Information Base (PIB) Attributes

phyCurrentChannel

Defines the current active channel for transmission, the range is limited to the available channels for 802.15.4.

phyChannelsSupported

An array describing which channels are supported by the current device.

phyTransmitPower

The transmit power of this unit

phyCCAMode

The Clear-Channel-Assessment CCA mode in WirelessHART is hard-coded to the value 2 as opposed to CCA in 802.15.4 which provides several modes.

phyCurrentPage

Channel page is not relevant for the 2.4GHz ISM band

phyMaxFrameDuration

This is the maximum number of symbols that can be transmitted in a frame

phySHRDuration

The duration of the synchronization header

phySymbolsPerOctet

The number of symbols per octet (byte)

Appendix C

Risk Assessment

In the following table we provide an overview of the risk assessment in terms of this project, the table includes a list of risks that we have identified as possible throughout the project and the severity and probability that it will occur. In addition, after the table, a more detailed description of each of the risks is provided along with a strategy for solving or minimizing the impact of the problem.

ID	Risk	Severity	Probability
R01	Unable to finish the project on time	HIGH	MEDIUM
R02	Implementation too slow	HIGH	HIGH
R03	Timing issues between entities	HIGH	MEDIUM
R04	Existing bugs in initial code base	LOW	MEDIUM

R01 By writing extensive and good documentation under way, we can reduce the implications of features which we are unable to fully implement making it easier to pick up for the next set of students to pick up this project.

R02 Put a lot of effort into making sure a time slot can be served within the requirements of the WirelessHART specification[29]. This in-

cludes function-testing existing algorithms against basic time constraints.

R03 The Gateway and nodes will run under different operating environments, possible issues will include clock drift and timer inaccuracies causing these two entities to drifts too quickly apart. There is not much we can do about this except change hardware.

R04 There is a fairly high probability that there will be bugs in the existing implementation and that we need to perform some debugging and bug fixing in order to get the implementation running smoothly. However, we can spend these opportunities to get to know the implementation and behavior of the existing code base.

Appendix D

Planning

Throughout this project we have utilized the unified process model (Figures D.1 and D.2) as an overall measure and model on how and where to focus our efforts. The unified process defines a set of phases in which we classify our work.

D.1 Inception

This is the smallest phase and consists in our case on getting familiar with previous work, related work in the field in addition to the problem area. In addition gaining an in-depth overview over the available protocols in the same area.

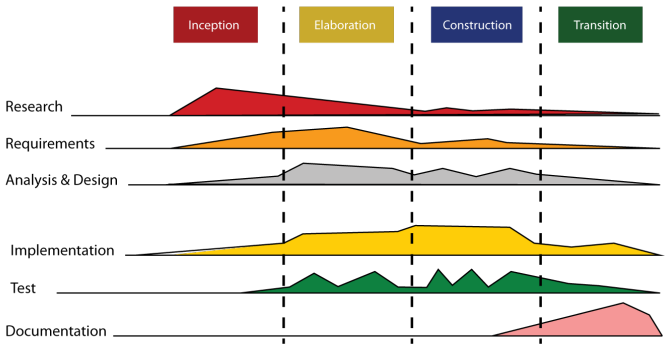


Figure D.1: Unified Process - Detail View

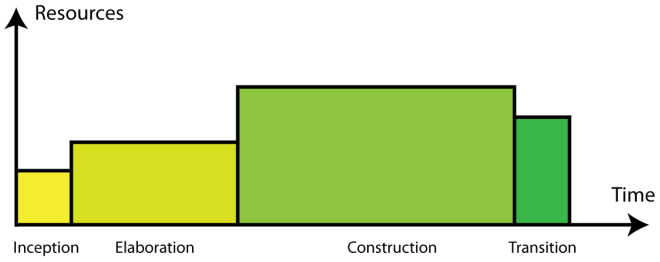


Figure D.2: Unified Process - Division of work

D.2 Elaboration

This is the primary design phase and the goal of this phase is to design and formulate use cases and the overall architecture, define functional and non-functional requirements. In addition with the knowledge we have gained through this and the previous phase we are able to identify and define risks.

D.3 Construction

This is the implementation phase where the project and the design described in phase 2 is put into practice. This is by far the longest and most challenging phase of this project.

D.4 Transition

The transition phase in the unified process model is usually when the software is deployed to the user. However, in our case, the transition phase is where we document, test and evaluate our work in order to provide a strong platform for the continuation of development.

Appendix E

Repository statistics

Throughout the project we have used Subversion in combination with Git for version control of our implementation and thesis. The below figures show different statistics on commits to the source code repository during the course of the project.

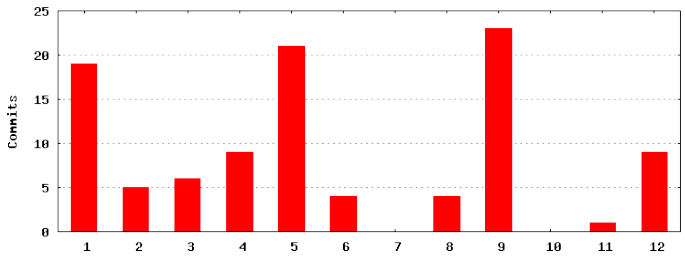


Figure E.1: Number of commits per month

Figure E.1 shows the number of commits in each month of the year. Since the project lasted from January 2011 to July 2012 January-July show the combined number of commits from this period in 2011 and 2012.

Figure E.2 displays the number of commits each year. As seen in the graph there are about the same amount of commits each year. Seeing that 30 credits of work were done in 2011 and 30 credits in 2012 this illustrates that the workload has been divided as expected over these two periods.

The last graph (Figure E.3) shows the commits to different branches of the project. The red line being the most important one goes a long

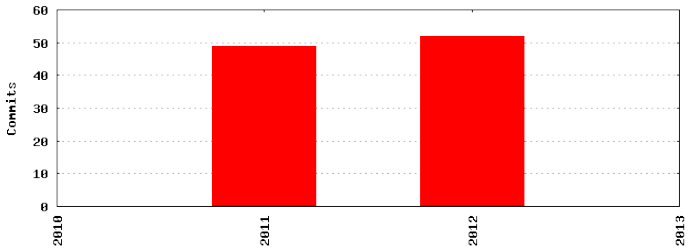


Figure E.2: Number of commits per year

way to illustrate where and when the effort has been made throughout the project. It matches quite well with the division of effort and labor in figure 8.5. The other lines may be disregarded as they are small branches of tests or features that have been discarded.

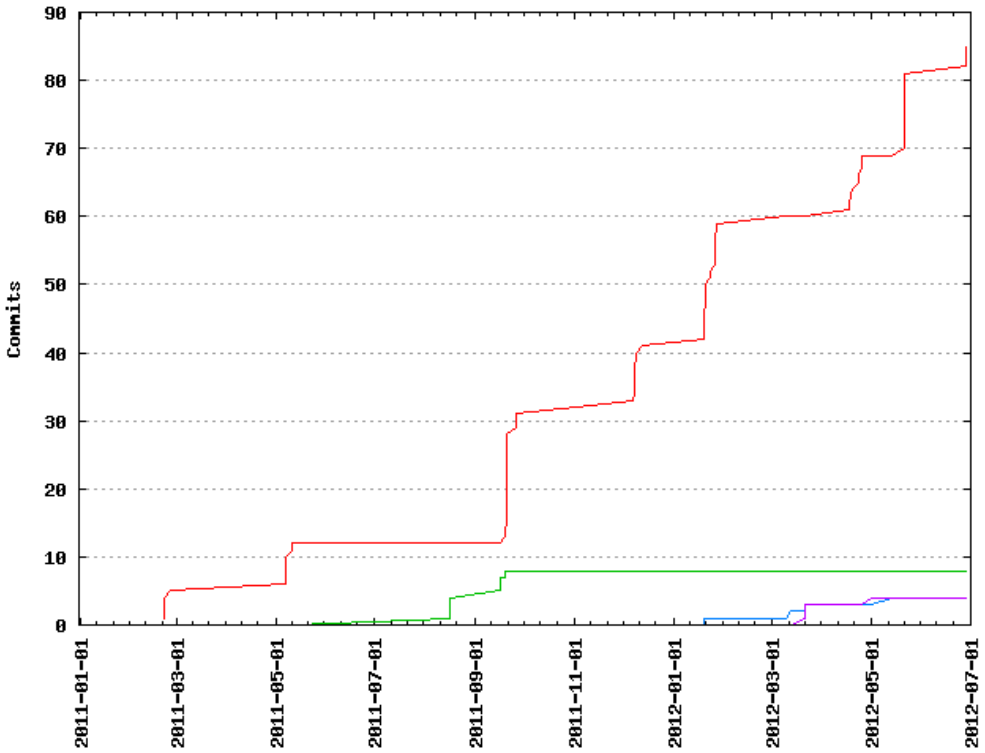


Figure E.3: Commit Activity - Each line represents a branch in our repository

Appendix F

API Reference

F.1 Header Files

F.1.1 wirelesshart_constants.h

This file contains MAC layer type definitions and data structures in addition to defining some basic WirelessHART constants.

F.1.2 wirelesshart_superframe.h

This file contains definitions directly related to the handling and constraints of WirelessHART super frames and slot timing requirements.

F.1.3 mac_api.h

This file contains function headers and definitions made available from the MAC layer for use by the network layer. This should be the only file that is included in the network layer application.

F.1.4 mac.h

This file contains function headers and definitions that the physical layer can use in order to notify the data link layer about events.

F.1.5 phy_api.h

This file contains function headers and definitions made available from the physical layer for use by the data link layer. This should be the only file that is included in the MAC layer.

F.1.6 mac_communication_tables.h

This file contains function headers and definitions for modifying the communication data structures such as links, nodes, neighbors etc. The functions defined in this file should only be used within the MAC layer itself and any functionality available in this file which is also required on higher layers should be available through local management service primitives as defined by the WirelessHART standard [29, 5.3.3].

F.1.7 mac_tdma_machine.h

This file contains function headers and definitions for invoking the TDMA state machine within the MAC layer. This file should not be referenced from any other layers.

F.1.8 mac_internal.h

This file contains function headers and definitions for internal data structures in the MAC layer in addition to function definitions for creating a variety of different data packets.

F.2 Wireless Gateway file listing

In this file listing, all files have been implemented during this project except the core functionality of GraphViz.

```
no
|-- uio
    |-- ifi
```

```
|-- wirelesshart
    |-- accesspoint
    |   |-- AccessPoint.java
    |   |-- dll
    |   |   |-- dllAPI.java
    |   |   |-- HIDRawUSBdll.java
    |   |-- HIDRawUSBAccessPoint.java
    |   |-- net
    |   |   |-- HIDRawUSBnet.java
    |   |   |-- netAPI.java
    |   |   |-- sec
    |   |       |-- HIDRawUSBsec.java
    |   |       |-- secAPI.java
    |   |-- pal
    |   |   |-- HIDAPRxRunner.java
    |   |   |-- HIDAPTx.java
    |   |   |-- HIDRawUSBpal.java
    |   |   |-- palAPI.java
    |   |-- pdu
    |   |   |-- AckDLPDU.java
    |   |   |-- AdvertiseDLPDU.java
    |   |   |-- DataDLPDU.java
    |   |   |-- DisconnectDLPDU.java
    |   |   |-- DLPDU.java
    |   |   |-- KeepAliveDLPDU.java
    |   |   |-- NPDU.java
    |   |   |-- PPDU.java
    |   |   |-- SPDU.java
    |   |-- SerialAccessPoint.java
    |   |-- session
    |   |   |-- HIDRawUSBses.java
    |   |   |-- sesAPI.java
    |   |-- statemachine
```

```
|   |   |-- StateMachine.java
|   |-- tdmaengine
|   |   |-- TDMAEngine.java
|   |-- TestAccessPoint.java
|   |-- tl
|       |-- HIDRawUSBtl.java
|       |-- tlAPI.java
|-- common
|   |-- Command.java
|   |-- Event.java
|   |-- StatisticsEvent.java
|   |-- StatisticsHandler.java
|-- gateway
|   |-- VirtualGatewayCommands.java
|   |-- VirtualGateway.java
|-- log
|   |-- LogFormatter.java
|   |-- LogHandler.java
|   |-- LurLogger.java
|-- main
|   |-- Gateway.java
|   |-- main1.ucd
|-- manager
|   |-- CircularLinkedList.java
|   |-- CircularList.java
|   |-- Link.java
|   |-- LinkType.java
|   |-- Neighbor.java
|   |-- NetworkManager.java
|   |-- SecurityManager.java
|   |-- SuperFrame.java
|-- ui
    |-- graphviz
```

```
|   |-- GraphViz.java
|   |-- Proba.java
|   |-- WHARTGraph.java
|-- GUIStatsRunner.java
|-- NetManShell.java
|-- NetworkManagerApplicationGUI.java
```

F.3 WirelessHART Field Device file listing

In the code for the WirelessHART Field Devices, all files which do not contain Atmel copyright comment notification at the top of the file has either been created or modified by us or Tegelsrud and Frøysadal.

```
.
|-- Include
|   |-- config.h
|   |-- ieee_const.h
|   |-- phy_api.h
|   |-- return_val.h
|   |-- test.h
|-- MAC
|   |-- Inc
|   |   |-- mac.h
|   |   |-- mac_api.h
|   |   |-- mac_communication_tables.h
|   |   |-- mac_create_dlpdu.h
|   |   |-- mac_internal.h
|   |   |-- mac_msg_const.h
|   |   |-- mac_tdma_machine.h
|   |   |-- wirelesshart_constants.h
|   |   |-- wirelesshart_superframe.h
|   |-- Src
```

```
|       |-- mac.c
|       |-- mac_api.c
|       |-- mac_communication_tables.c
|       |-- mac_create_dlpdu.c
|       |-- mac_link_scheduler.c
|       |-- mac_misc.c
|       |-- mac_recv_engine.c
|       |-- mac_tdma_machine.c
|       |-- mac_xmit_engine.c
|-- NAL
|   |-- Inc
|   |   |-- nal.h
|   |   |-- nal_api.h
|   |   |-- nal_operation_state.h
|   |-- Src
|       |-- nal.c
|       |-- nal_api.c
|       |-- nal_operation_state.c
|-- PAL
|   |-- Inc
|   |   |-- avrtypes.h
|   |   |-- pal.h
|   |   |-- pal_config.h
|   |   |-- pal_internal.h
|   |   |-- pal_timer.h
|   |-- Src
|       |-- pal.c
|       |-- pal_board.c
|       |-- pal_irq.c
|       |-- pal_timer.c
|       |-- pal_trx_access.c
|-- PHY
|   |-- Inc
```

```
|   |-- Src
|-- Resources
|   |-- Buffer_Management
|   |   |-- Inc
|   |   |   |-- bmm.h
|   |   |-- Src
|   |       |-- bmm.c
|   |-- List_Management
|   |   |-- Inc
|   |   |   |-- list_manager.h
|   |   |-- Src
|   |       |-- list_manager.c
|   |-- Queue_Management
|       |-- Inc
|       |   |-- qmm.h
|       |-- Src
|           |-- qmm.c
|-- Src
    |-- main.c
    |-- phy_api.c
```

F.4 15dot4-tools file listing

In this file listing, the only files that have been modified during this project are marked with an asterisk (*).

```
.
|-- common
|   |-- avr
|   |   |-- avr_serial.c
|   |   |-- avr_serial.h
|   |   |-- avr_timer.c
|   |   |-- avr_timer.h
```

```
|   |   |-- hal_avr.c
|   |   |-- hal_avr.h
|   |-- mac
|   |   |-- mac.c
|   |   |-- mac.h
|   |   |-- mac_event.c
|   |   |-- mac_event.h
|   |   |-- mac_framequeue.c
|   |   |-- mac_framequeue.h
|   |   |-- mac_scan.c
|   |   |-- mac_scan.h
|   |   |-- mac_sniffer.c*
|   |   |-- mac_sniffer.h*
|   |   |-- rum_types.h
|   |   |-- stdboolrum.h
|   |   |-- system.h
|   |-- radio
|   |   |-- at86rf212_registermap.h
|   |   |-- at86rf23x_registermap.h
|   |   |-- hal.c
|   |   |-- hal.h
|   |   |-- radio.c*
|   |   |-- radio.h
|   |-- serial
|   |   |-- serial.c
|   |   |-- serial.h
|   |-- timer
|   |   |-- timer.c
|   |   |-- timer.h
|   |-- usb
|       |-- eem
|       |   |-- eem.h
|       |   |-- eem_avr.c
```

```
|       |-- hal
|       |   |-- avr
|       |   |   |-- conf_usb.h
|       |   |   |-- config.h
|       |   |   |-- highlevel
|       |   |   |   |-- usb_specific_request.c
|       |   |   |   |-- usb_specific_request.h
|       |   |   |   |-- usb_standard_request.c
|       |   |   |   |-- usb_standard_request.h
|       |   |   |-- lowlevel
|       |   |   |   |-- pll_drv.h
|       |   |   |   |-- usb_drv.c
|       |   |   |   |-- usb_drv.h
|       |   |   |-- usb_commun.h
|       |   |   |-- usbtask
|       |   |       |-- usb_device_task.c
|       |   |       |-- usb_device_task.h
|       |   |       |-- usb_task.c
|       |   |       |-- usb_task.h
|       |   |-- compiler.h
|       |   |-- hal_usb.h
|       |-- hid
|       |   |-- TcfTransactionLog.csv
|       |   |-- hid.c*
|       |   |-- hid.h*
|       |   |-- hid_avr.c
|       |-- massstorage
|       |   |-- MSDDStateMachine.c
|       |   |-- MSDDriver.c
|       |   |-- MSDIOFifo.c
|       |   |-- MSDLun.c
|       |   |-- MSDLun.h
|       |   |-- SBC.h
```

```
|       |   |-- SBCMethods.c
|       |   |-- SBCMethods.h
|       |   |-- msd.h
|       |-- rndis
|           |-- ndis.h
|           |-- rndis.c
|           |-- rndis.h
|           |-- rndis_avr.c
|-- sniffer
    |-- host
    |   |-- CompositeAtmelRNDIS.inf
    |   |-- commandline_app
    |       |-- Makefile
    |       |-- main.c
    |-- peripherial
        |-- application.c
        |-- avr_gcc_build
        |   |-- Makefile
        |   |-- sniffer.noboot.bin
        |-- main.c
        |-- usb_code
            |-- debug_task.c
            |-- debug_task.h
            |-- ethusb_task.c*
            |-- ethusb_task.h
            |-- usb.h
            |-- usb_descriptors_avr.c
            |-- usb_descriptors_avr.h
```